SOFTWARE TESTING ASSISTANT

NSQF LEVEL - 4.5

Volume II of II

TRADE PRACTICAL

SECTOR: IT & ITES

(As per revised syllabus 2023 - 1200 hrs)

© NIMI
NOT TO BE REPUBLISHED

Directorate General of Training

DIRECTORATE GENERAL OF TRAINING
MINISTRY OF SKILL DEVELOPMENT & ENTREPRENEURSHIP
GOVERNMENT OF INDIA

NATIONAL INSTRUCTIONAL MEDIA INSTITUTE, CHENNAI

Post Box No. 3142, CTI Campus, Guindy, Chennai - 600 032

Sector : **IT & ITES**

Duration : **1 Year**

Trade : **Software Testing Assistant - Trade Practical (Vol II of II) - NSQF Level - 4.5 (Revised 2023)**

Developed & Published by

National Instructional Media Institute
Post Box No.3142
Guindy, Chennai - 600 032
INDIA
Email: chennai-nimi@nic.in
Website: www.nimi.gov.in

Copyright © 2025 National Instructional Media Institute, Chennai

First Edition: March 2025 Copies: 1,000

© NIMI
NOT TO BE REPUBLISHED

Rs.345/-

All rights reserved.

No part of this publication can be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording or any information storage and retrieval system, without permission in writing from the National Instructional Media Institute, Chennai.

FOREWORD

The Government of India has set an ambitious target of imparting skills to 30 crores people, one out of every four Indians to help them secure jobs as part of the National Skills Development Policy. Industrial Training Institutes (ITIs) play a vital role in this process especially in terms of providing skilled manpower. Keeping this in mind, and for providing the current industry relevant skill training to Trainees, ITI syllabus has been recently updated with the help of Mentor Councils comprising various stakeholder's viz. Industries, Entrepreneurs, Academicians and representatives from ITIs.

The National Instructional Media Institute (NIMI), Chennai has now come up with instructional material to suit the revised curriculum for **Software Testing Assistant - Trade Practical (Vol II of II) - NSQF Level - 4.5 (Revised 2023) in IT & ITES Sector.** The NSQF Level - 4.5 (Revised 2023) Trade Practical will help the trainees to get an international equivalency standard where their skill proficiency and competency will be duly recognized across the globe and this will also increase the scope of recognition of prior learning. NSQF Level - 4.5 (Revised 2023) trainees will also get the opportunities to promote life long learning and skill development. I have no doubt that with NSQF Level - 4.5 (Revised 2023) the trainers and trainees of ITIs, and all stakeholders will derive maximum benefits from these IMPs and that NIMI's effort will go a long way in improving the quality of Vocational training in the country.

The Director General, Executive Director & Staff of NIMI and members of Media Development Committee deserve appreciation for their contribution in bringing out this publication.

Jai Hind

© NIMI
NOT TO BE REPUBLISHED

ATUL KUMAR TIWARI, *I.A.S,*
Secretary,
Ministry of Skill Development & Entrepreneurship,
Government of India.

March 2025
New Delhi - 110 001

PREFACE

The National Instructional Media Institute (NIMI) was established in 1986 at Chennai by then Directorate General of Employment and Training (D.G.E & T), Ministry of Labour and Employment, (now under Ministry of Skill Development and Entrepreneurship) Government of India, with technical assistance from the Govt. of the Federal Republic of Germany. The prime objective of this institute is to develop and provide instructional materials for various trades as per the prescribed syllabi under the Craftsman and Apprenticeship Training Schemes.

The instructional materials are created keeping in mind, the main objective of Vocational Training under NCVET/NAC in India, which is to help an individual to master skills to do a job. The instructional materials are generated in the form of Instructional Media Packages (IMPs). An IMP consists of Theory book, Practical book, Test and Assignment book, Instructor Guide, Audio Visual Aid (Wall charts and Transparencies) and other support materials.

The trade practical book consists of series of exercises to be completed by the trainees in the workshop. These exercises are designed to ensure that all the skills in the prescribed syllabus are covered. The trade theory book provides related theoretical knowledge required to enable the trainee to do a job. The test and assignments will enable the instructor to give assignments for the evaluation of the performance of a trainee. The wall charts and transparencies are unique, as they not only help the instructor to effectively present a topic but also help him to assess the trainee's understanding. The instructor guide enables the instructor to plan his schedule of instruction, plan the raw material requirements, day to day lessons and demonstrations.

In order to perform the skills in a productive manner instructional videos are embedded in QR code of the exercise in this instructional material so as to integrate the skill learning with the procedural practical steps given in the exercise. The instructional videos will improve the quality of standard on practical training and will motivate the trainees to focus and perform the skill seamlessly.

IMPs also deals with the complex skills required to be developed for effective team work. Necessary care has also been taken to include important skill areas of allied trades as prescribed in the syllabus.

The availability of a complete Instructional Media Package in an institute helps both the trainer and management to impart effective training.

The IMPs are the outcome of collective efforts of the staff members of NIMI and the members of the Media Development Committees specially drawn from Public and Private sector industries, various training institutes under the Directorate General of Training (DGT), Government and Private ITIs.

NIMI would like to take this opportunity to convey sincere thanks to the Directors of Employment & Training of various State Governments, Training Departments of Industries both in the Public and Private sectors, Officers of DGT and DGT field institutes, proof readers, individual media developers and coordinators, but for whose active support NIMI would not have been able to bring out this materials.

Chennai - 600 032 **EXECUTIVE DIRECTOR**

© NIMI NOT TO BE REPUBLISHED

ACKNOWLEDGEMENT

National Instructional Media Institute (NIMI) sincerely acknowledges with thanks for the co-operation and contribution extended by the following Media Developers and their sponsoring organisations to bring out this Instructional Material **(Trade Practical)** for the trade of **Software Testing Assistant (Vol II of II)** (NSQF Level - 4.5) (Revised 2023) under **IT & ITES** Sector for ITIs.

MEDIA DEVELOPMENT COMMITTEE MEMBERS

Shri. Sarbojit Neogi	-	Training Officer, NSTI (W) Kolkata.
Shri. C. Jeyaprakash	-	MDC Member, NIMI, Chennai - 32

NIMI CO-ORDINATORS

Shri. Nirmalya Nath	-	Deputy Director, NIMI- Chennai - 32.
Shri. S. Gopala Krishnan	-	Assistant Manager, NIMI, Chennai - 32

NIMI records its appreciation for the Data Entry, CAD, DTP operators for their excellent and devoted services in the process of development of this Instructional Material.

NIMI also acknowledges with thanks the invaluable efforts rendered by all other NIMI staff who have contributed towards the development of this Instructional Material.

NIMI is also grateful to everyone who has directly or indirectly helped in developing this Instructional Material.

© NIMI NOT TO BE REPUBLISHED

INTRODUCTION

TRADE PRACTICAL

The trade practical manual is intended to be used in workshop . It consists of a series of practical exercises to be completed by the trainees during the course of the **Software Testing Assistant (Vol II of II)** Trade supplemented and supported by instructions/ informations to assist in performing the exercises. These exercises are designed to ensure that all the skills in compliance with NSQF Level - 4.5 (Revised 2023) syllabus are overed.

The manual is divided into four modules.

Module 10 - Manual and Principles of Testing

Module 11 - Automating Test Execution

Module 12 - Windows Automated Testing Tool

Module 13 - Web Automated Testing Tool & Comparison

The skill training in the shop floor is planned through a series of practical exercises centered around some practical project. However, there are few instances where the individual exercise does not form a part of project.

While developing the practical manual a sincere effort was made to prepare each exercise which will be easy to understand and carry out even by below average trainee. However the development team accept that there is a scope for further improvement. NIMI, looks forward to the suggestions from the experienced training faculty for improving the manual.

TRADE THEORY

The manual of trade theory consists of theoretical information for the course of the **Software Testing Assistant (Vol II of II) - NSQF Level - 4.5 (Revised 2023)** in **IT & ITES**.The contents are sequenced according to the practical exercise contained in the manual on Trade practical. Attempt has been made to relate the theoretical aspects with the skill covered in each exercise to the extent possible. This co-relation is maintained to help the trainees to develop the perceptional capabilities for performing the skills.

The Trade theory has to be taught and learnt along with the corresponding exercise contained in the manual on trade practical. The indicating about the corresponding practical exercise are given in every sheet of this manual.

It will be preferable to teach/learn the trade theory connected to each exercise atleast one class before performing the related skills in the shop floor. The trade theory is to be treated as an integrated part of each exercise.

The material is not the purpose of self learning and should be considered as supplementary to class room instruction.

© NIMI NOT TO BE REPUBLISHED

CONTENTS

Exercise No.	Title of the Exercise	Learning Outcome	Page No.
	Module 10: Manual and Principles of Testing		
1.10.60	Unit Testing		1
1.10.61	Alpha & Beta Testing		4
1.10.62	Regression vs Retesting		7
1.10.63	White box testing	10	9
1.10.64	Black box testing		11
1.10.65	White box v/s Black box		14
1.10.66	Verification & Validation		17
1.10.67	Acceptance Testing		19
1.10.68	Non-Functional Testing		21
1.10.69	Usability Testing		23
1.10.70	Stress Testing	10	25
1.10.71	Load Testing		27
1.10.72	Performance Testing		30
1.10.73	Diff. b/w above 3		33
	Module 11: Automating Test Execution		
1.11.74	The benefits of automated testing		35
1.11.75	Understanding the testing process		38
1.11.76	Exploring the WinRunner Window		41
1.11.77	How does WinRunner identify GUI objects		44
1.11.78	Spying on GUI map mode		47
1.11.79	Choosing a GUI map mode		49
1.11.80	Using the rapid test script wizard		51
1.11.81	Choosing a record mode		53
1.11.82	Recording a context sensitive test		55
1.11.83	Understanding the text script		56
1.11.84	Recording in analogy mode		58
1.11.85	Running the test		60
1.11.86	Analysing test results		62
1.11.87	Recording tips		64
1.11.88	When should you synchronize?		67
1.11.89	Creating a test		70
1.11.90	Changing the synchronization setting	11	73
1.11.91	Identifying a synchronization problem		75
1.11.92	Synchronizing the test		77
1.11.93	Running the synchronized test		80
1.11.94	How do you check a bitmap?		84
1.11.95	Adding bitmap checkpoints to a test script		86
1.11.96	Viewing expected results		88
1.11.97	Running the test on a new version		91
1.11.98	Bitmap checkpoint tips		93

© NIMI NOT TO BE REPUBLISHED

Exercise No.	Title of the Exercise	Learning Outcome	Page No.
1.11.99	How do you program tests with TSL		95
1.11.100	Recording a basic test script		97
1.11.101	Using the function generator to insert functions		99
1.11.102	Adding logic to the test scrip		102
1.11.103	Understanding TL-step		107
1.11.104	Debugging the test script		109
1.11.105	Running the test on a new version		113
1.11.106	How do you create data - driven tests?		116
1.11.107	Converting your test to a data driven test		118
1.11.108	Adding data to the data table		121
1.11.109	Adjusting the script with regular information		123
1.11.110	Running the test and analysing result		126
1.11.111	Data driven testing tips		129
1.11.112	How do you read text from an application?		132
1.11.113	Reading text from an application	11	134
1.11.114	Teaching fonts to win runner		136
1.11.115	Verifying text		138
1.11.116	Running the test on a new version		140
1.11.117	Text checkpoint tips		142
1.11.118	What is a batch test?		144
1.11.119	Programming a batch test		147
1.11.120	Running the batch test on version IB		149
1.11.121	Analysing the batch test results		151
1.11.122	Batch test tips		154
1.11.123	What happens when the user interface changes?		157
1.11.124	Editing object descriptions in the GUI map adding GUI objects to the GUI map		158
1.11.125	Updating the GUI map with the run wizard		159
	Module 12: Windows Automated Testing Tool		
1.12.126	Load test planning		160
1.12.127	The LoadRunner controller at a glance		162
1.12.128	Creating a scenario		164
1.12.129	Using rendezvous points		166
1.12.130	Configuring a scenario		168
1.12.131	Configuring a host	12	170
1.12.132	Preparing to run a scenario		172
1.12.133	Managing scenarios using test director		174
1.12.134	Running a scenario		176
1.12.135	Online monitoring		178
1.12.136	Runtime and transaction online monitors		180

© NIMI NOT TO BE REPUBLISHED

Exercise No.	Title of the Exercise	Learning Outcome	Page No.
1.12.137	Resource monitoring		182
1.12.138	Web performance monitors		184
1.12.139	Network monitoring		186
1.12.140	Understanding load runner analysis		188
1.12.141	Exporting analysis data	12	190
1.12.142	Analysing scenario activity		191
1.12.143	Analysing scenario performance		193
1.12.144	Cross scenario analysis		195
1.12.145	Web user graphs		197
	Module 13: Web Automated Testing Tool & Comparison		
1.13.146	Installing the IDE		199
1.13.147	Opening the IDE		203
1.13.148	IDE Features		204
1.13.149	Menu bar, toolbar, case pane		205
1.13.150	Log / reference / UI- element rollup pane		206
1.13.151	Log, reference, I - element and rollup		208
1.13.152	Building test cases		210
1.13.153	Recording		211
1.13.154	Adding verifications and asserts with the context menu		212
1.13.155	Editing, insert command, table view		214
1.13.156	Source view, insert comment, table view, source view, edit a command or comment		216
1.13.157	Table view, source view, opening and saving a test case		219
1.13.158	Running test cases		221
1.13.159	Using base URL to run test cases in different domains	13	223
1.13.160	Selenium commands -”Selenese”		225
1.13.161	Script syntax		227
1.13.162	Test suites		229
1.13.163	Commonly used selenium commands		231
1.13.164	Verifying page elements		233
1.13.165	Assertion or Verification?		235
1.13.166	Verify text present, verify element present		237
1.13.167	Verify text		238
1.13.168	Location elements		239
1.13.169	Location by identifier, location by id		240
1.13.170	Location by name, location by X-path		241
1.13.171	Location hyperlinks by link text		243
1.13.172	Location by DOM, Location by CSS		244
1.13.173	Implicit locators		246
1.13.174	Matching text patterns		248

© NIMI NOT TO BE REPUBLISHED

Exercise No.	Title of the Exercise	Learning Outcome	Page No.
1.13.175	Globbing patterns, regular expression patterns, exact patterns		250
1.13.176	The “And Wait” Commands		251
1.13.177	The wait for commands in AJAX applications		252
1.13.178	Sequence of evaluation and flow control		253
1.13.179	Store commands and selenium variables		257
1.13.180	Store element present, store text, store evaluation		259
1.13.181	JavaScript and selenese parameters		261
1.13.182	JavaScript usages with script parameters		263
1.13.183	JavaScript usage with non-script parameters		265
1.13.184	Echo - the selenese print commands	13	267
1.13.185	Alert, popups and multiple windows		268
1.13.186	Alerts, Confirmations		270
1.13.187	Debugging		272
1.13.188	Breakpoint and start points		274
1.13.189	Stepping through a test case		276
1.13.190	Find button		278
1.13.191	Page source for debugging		280
1.13.192	Locator assistance		282
1.13.193	Writing a test suite		285
1.13.194	User extensions		287
1.13.195	Format		289
1.13.196	Executing selenium-IDE tests on different browsers		291
1.13.197	Troubleshooting		293

© NIMI NOT TO BE REPUBLISHED

LEARNING / ASSESSABLE OUTCOME

On completion of this book you shall be able to

S.No.	Learning Outcome	Exercise No.
10	**Apply manual testing techniques in Software Testing. (NOS: SSC/N1303)**	**1.10.60 - 1.10.73**
11	**Perform automatic test execution using Windows automated software testing tool WinRunner. (NOS:SSC/N1302)**	**1.11.74 - 1.11.125**
12	**Perform automatic test execution using Windows automated software testing tool LoadRunner. (NOS:SSC/N1302)**	**1.12.126 - 1.12.145**
13	**Perform automatic test execution using Web automated software testing tool Selenium IDE.(NOS:SSC/N1302)**	**1.13.146 - 1.13.197**

Software Testing Assistant - Manual and Principles of Testing

Unit Testing

Objective: At the end of this exercise you shall be able to
- **unit testing ensures individual components function correctly.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with frameworks	- as reqd.

PROCEDURE

Unit testing: Step-by-step guide with output

Unit Testing is the process of testing individual units or components of an application (like functions, methods, or classes) in isolation to ensure that they perform as expected. It helps verify that each part of the code works correctly before integrating it with other components.

Unit testing is typically done by developers, but testers may also get involved in validating individual functions. It is often automated, but it can also be performed manually in some cases.

Below is a step-by-step guide for performing Unit Testing with outputs, including both automated and manual approaches.

Step 1: Identify the unit to be tested

- **Objective:** Select the specific unit or component that will be tested.
- **Action:** A "unit" is the smallest testable part of the application, such as a function, method, or class.
- **Example:** Testing a function calculateTotalPrice() that calculates the total price for a given order.

Step 2: Understand the expected behaviour

- **Objective**: Understand how the unit should behave, including inputs, outputs, edge cases, and error handling.
- **Action:** Review the requirements or specifications to define the expected output for valid and invalid inputs.

Example

- **Unit to test:** calculateTotalPrice(price, quantity, discount)
- **Inputs:** price (float), quantity (int), discount (float)
- **Expected outputs:** The total price after applying the discount.
- **Edge cases:** Invalid input like negative values, zero quantity, or missing discount.

Step 3: Write test cases

- **Objective:** Create test cases to validate that the unit behaves as expected.
- **Action:** Test cases should cover
  - Normal cases (valid inputs).
  - Edge cases (boundary conditions).
  - Error cases (invalid inputs or unexpected behaviour).

Example test cases

i **Test case 1 (Normal case)**
- **Input:** Price = $100, Quantity = 3, Discount = 10%
- **Expected output:** $270 (Total price after discount)

ii **Test case 2 (Zero quantity)**
- **Input:** Price = $100, Quantity = 0, Discount = 10%
- **Expected output:** $0 (Total price is zero because quantity is zero)

iii **Test case 3 (Negative price)**
- **Input:** Price = -$100, Quantity = 3, Discount = 10%
- **Expected output:** Error (Invalid input, should handle negative price)

Step 4: Set up the test environment

- **Objective:** Ensure that the environment is configured properly for running the unit test.
- **Action:** Set up any necessary data, mock objects, or environments needed to test the unit in isolation. If using automated unit testing, ensure the test framework (e.g., JUnit, pytest) is properly configured.

Example

- Prepare test data for the calculateTotalPrice() function.
- Ensure the function is correctly imported into the test file.

Step 5: Execute the unit test

- **Objective:** Run the unit test to check whether the unit behaves as expected.

© NIMI NOT TO BE REPUBLISHED

- **Action:** Run the test cases and document the results.
  - Compare the expected result with the actual result.
  - Mark the test case as Pass or Fail based on the comparison.

Automated unit testing example (using Python and pytest)

Test code (Fig 1)

Fig 1

```python
def calculate_total_price(price, quantity, discount):
    if price < 0 or quantity < 0:
        raise ValueError("Price and Quantity must be non-negative")
    total = price * quantity
    total -= total * (discount / 100)
    return total

def test_calculate_total_price():
    # Test Case 1
    assert calculate_total_price(100, 3, 10) == 270  # Normal Case

    # Test Case 2
    assert calculate_total_price(100, 0, 10) == 0  # Zero Quantity

    # Test Case 3
    try:
        calculate_total_price(-100, 3, 10)  # Invalid Input
    except ValueError as e:
        assert str(e) == "Price and Quantity must be non-negative"  # Error Handling
```

© NIMI NOT TO BE REPUBLISHED

Step 6: Record the test results

- **Objective:** Document the outcome of the unit test execution.
- **Action:** Record whether each test case passed or failed. If it fails, note any defects or unexpected behaviours.

Example test execution results

Test Case ID	Description	Expected Output	Actual Output	Status
TC_001	Calculate total price with valid inputs	270	270	Pass
TC_002	Calculate total price with zero quantity	0	0	Pass
TC_003	Calculate total price with negative price	Error (ValueError)	Error (ValueError)	Pass

Step 7: Report defects (If any)

- **Objective:** If a test case fails, create a defect report.
- **Action:** Log the defect in a defect management tool (e.g., Jira, Bugzilla). Include the following:
  - Steps to reproduce the error.
  - Expected vs Actual result.
  - Severity (Critical, Major, Minor).

Example defect report

- **Defect ID:** DEF_001
- **Test Case:** TC_003
- **Issue:** Invalid input handling for negative price.
- **Steps to reproduce**
  - Call calculate_total_price(-100, 3, 10).
  - Check error handling for negative price.

- **Expected result:** Error with a clear message for invalid input.
- **Actual result:** Error message is correctly shown.
- **Severity:** Major

Step 8: Fix the issues (If any)

- **Objective:** Ensure that any issues identified in the unit test are fixed.
- **Action:** Developers address the identified defect(s) and fix the unit or its implementation. After the fix, re-run the unit tests to verify that the issue has been resolved.

Example fix

- The developer fixes the issue where negative prices were allowed in the calculation. The function is updated to handle such cases and throw a ValueError.

Step 9: Retest and verify

- **Objective:** Confirm that the fix works and that no other functionality has been broken.
- **Action:** Re-run the unit tests, especially the failed test cases, and verify that all tests pass.

Retest results

Test Case ID	Description	Expected Output	Actual Output	Status
TC_001	Calculate total price with valid inputs	270	270	Pass
TC_002	Calculate stotal price with zero quantity	0	0	Pass
TC_003	Calculate total price with negative price	Error (ValueError)	Error (ValueError)	Pass

— — — — — —

© NIMI
NOT TO BE REPUBLISHED

Alpha & Beta Testing

Objective: At the end of this exercise you shall be able to
- **alpha and beta testing identify issues pre-release.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with test management andm testing platforms	- as reqd.

PROCEDURE

Alpha and beta testing: Step-by-step guide with output

Alpha and Beta testing are crucial stages in the software development lifecycle, allowing organizations to identify issues before releasing the product to the public.

- Alpha Testing is an internal phase, typically done by the developers and QA teams to catch critical bugs.
- Beta Testing is a stage where the software is tested by a select group of external users, providing valuable feedback from real-world use.

Let's break down the step-by-step process for both Alpha Testing and Beta Testing with output examples.

Alpha testing: Step-by-step process

Alpha Testing is the first phase of testing and is generally performed by an in-house team of developers, QA testers, and other stakeholders before releasing it to external testers or customers.

Step 1: Prepare the test environment

- **Objective:** Set up an environment that mimics the real-world user environment.
- **Action**
  - Set up the necessary infrastructure (test servers, network configurations, etc.).
  - Ensure that the software is in a pre-release stage.
  - The development team should be ready to address any issues found during testing.

Example: Set up the application on a staging server that mimics the production environment.

Step 2: Define alpha test scenarios

- **Objective:** Define the test scenarios based on software requirements.
- **Action**
  - Review the functional specifications, user stories, or requirements documents.
  - Create test cases focusing on core functionality, features, and integration points.
  - Test cases should cover both expected and edge-case scenarios.

Example test cases

i **Login:** Test login with valid/invalid credentials.

ii **Search:** Test product search functionality with various keywords.

iii **Checkout:** Test the checkout process for an order, including payment methods.

iv **Error handling:** Verify that appropriate error messages are displayed for invalid inputs.

Step 3: Execute test cases

- **Objective:** Run the defined test cases to identify any bugs or issues.
- **Action**
  - Execute the test cases manually or using automated test scripts.
  - Capture actual outputs and compare them with the expected outputs.

Example test case execution

Test Case	Expected Outcome	Actual Outcome	Status
Test Case 1 (Login)	Login with valid credentials, successful login	Successful login	Pass
Test Case 2 (Search)	Product found for keyword "laptop"	Product found	Pass
Test Case 3 (Checkout)	Order placed successfully	Error during payment processing	Fail

© NIMI NOT TO BE REPUBLISHED

Step 4: Report bugs and issues

- **Objective:** Log any defects found during testing.
- **Action**
  - Document the issues in a bug tracking tool (e.g., Jira, Bugzilla).
  - Provide steps to reproduce, expected vs. actual results, and severity of each defect.

Example bug report

- **Bug ID:** BUG-001
- **Test case:** Test Case 3 (Checkout)
- **Description:** Error during payment processing.
- **Steps to Reproduce**
  i Add items to the cart.
  ii Proceed to checkout.
  iii Enter payment details and submit.
- **Expected result:** Payment should be processed successfully.
- **Actual result:** Payment processing fails with an error message.
- **Severity:** Major

Step 5: Fix issues and retest

- **Objective:** Resolve any defects and ensure the software is stable.
- **Action**
  - Developers fix the reported bugs.
  - The QA team re-tests the fixed issues.
  - The process repeats until most critical bugs are resolved.

Retesting example

- **Bug:** Payment processing failure.
- **Fix:** The development team identifies an issue with the payment gateway integration and fixes it.
- **Test result:** After the fix, the payment processing test passes.

Step 6: Alpha testing completion

- **Objective:** Ensure all major bugs are fixed and the software is ready for external testing.
- **Action:** Once critical defects are resolved, the software is considered ready for Beta Testing.

Beta testing: Step-by-step process

Beta Testing is the phase where the software is tested by external users who are not part of the development team. Beta testers provide feedback from real-world scenarios, which is valuable for finalizing the product.

Step 1: Select beta testers

- **Objective:** Choose a group of external users to participate in testing.
- **Action**
  - Select a group of users who represent the target audience.
  - The group could be a closed set (invited) or open (volunteers).
  - Testers should have different levels of technical knowledge, if possible.

Example: Invite 50 external users (some experienced, some novice) to test the e-commerce app.

Step 2: Provide instructions for beta testing

- **Objective:** Ensure that testers understand how to use the software and provide feedback.
- **Action**
  - Send an introductory email to the testers.
  - Provide them with documentation on how to install and use the application.
  - Provide a feedback form or channel for them to report issues.

Example: Provide a user guide and a feedback form with questions such as:

- Was the software easy to use?
- Did you encounter any bugs or errors?
- What features do you think need improvement?

Step 3: Collect feedback

- **Objective:** Gather valuable feedback and bug reports from testers.
- **Action**
  - Ask beta testers to use the software for a specific period (e.g., two weeks).
  - Collect feedback on usability, performance, and any issues.

Example feedback

i **Usability:** “The product search feature is very slow on mobile devices.”

ii **Bug report:** “The app crashes when I try to apply a coupon code on the checkout page.”

iii **Feature request:** “It would be great to have a ‘Save for Later’ option in the cart.”

Step 4: Analyze feedback and report bugs

- **Objective:** Review the feedback and bugs provided by testers.
- **Action**
  - Organize the feedback into categories: bugs, performance issues, usability feedback, and feature requests.
  - Log bugs and issues in the defect management tool.

© NIMI NOT TO BE REPUBLISHED

Example bug report

- **Bug ID:** BUG-002
- **Test case:** Checkout Process
- **Description:** App crashes when applying coupon code.
- **Steps to reproduce**
  - i Add items to the cart.
  - ii Go to checkout.
  - iii Apply a coupon code.
- **Expected result:** Coupon code should be applied without crashing.
- **Actual result:** App crashes.
- **Severity:** Critical

Step 5: Fix issues and update the software

- **Objective:** Resolve the issues identified during Beta Testing.
- **Action**
  - Developers fix the bugs and address feature requests.
  - New versions or patches are released for the testers to validate the fixes.

Example fix

- **Bug:** App crashing when applying coupon code.
- **Fix:** Developers fix the issue by addressing a bug in the coupon code logic.
- **Test result:** After the fix, the coupon code applies correctly without causing a crash.

Step 6: Finalize product for release

- **Objective:** Make the product ready for the general public release.
- **Action**
  - After fixing the bugs and incorporating feedback, the product is finalized.
  - A final version of the software is released to the public.

Output from alpha & beta testing

1 Alpha testing output

- Test Cases Executed
- Bugs Reported (e.g., Checkout payment issue)
- Fixes Applied
- Test Status: Stable enough for Beta Testing

2 Beta testing output

- Beta Testers' Feedback (e.g., performance issues, usability feedback)
- Bugs Reported (e.g., app crash during coupon code application)
- Feature Requests (e.g., "Save for Later" feature)
- Fixes Implemented.

© NIMI NOT TO BE REPUBLISHED

Regression vs Retesting

Objective: At the end of this exercise you shall be able to
- **regression tests overall functionality retesting verifies fixes.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with automated testing tools	- as reqd.

PROCEDURE

In Manual Testing, Regression Testing and Retesting are two important testing techniques, but they have distinct purposes and processes. Let me break them down step by step

1 Regression testing

- **Purpose:** To verify that recent code changes, such as bug fixes, new features, or enhancements, have not negatively impacted existing functionality in the application.

Step-by-step process for regression testing

i Identify changes in code: Understand what parts of the application have been modified, including bug fixes, new features, or performance improvements.

ii Select test cases: Based on the changes, select test cases that cover the impacted areas, as well as critical functionalities of the application that could potentially be affected.

- **Example:** If a new "Forgot Password" feature is added, tests for the login, user profile, and password management areas might be selected.

iii Execute test cases: Run the selected test cases on the application to ensure that the existing functionalities are still working correctly.

- **Example:** Execute test cases for login, user registration, etc., even if those areas are not directly changed.

iv Compare results with expected outcomes: Verify whether the features work as they did before the changes.

v Report results

- **If pass:** Document that the system's existing functionality is unaffected.
- **If fail:** Report defects, which indicate that the new changes have introduced issues in previously working areas.

Output of regression testing

- **Pass:** The test cases that were executed work as expected, confirming no unintended side effects from the code changes.
- **Fail:** The test cases fail, indicating that the changes caused a regression or issue in the previously stable functionality.

2 Retesting

- **Purpose:** To verify that a specific defect or bug that was fixed has been resolved by executing the same test case that initially failed.

Step-by-step process for retesting

i Identify the fixed defect: Understand the nature of the defect or bug that was reported and fixed.

ii Select the same test case: Use the exact same test case that previously failed because of the defect, as well as the same test data.

- **Example:** If a defect was that a user couldn't log in due to incorrect password handling, the same login test case will be used.

iii Execute the test case: Run the selected test case on the application to confirm that the defect has been fixed and that the expected results are now achieved.

iv Compare results with expected outcomes: Check if the issue that caused the bug has been resolved.

v Report results

- **If pass:** Document that the defect has been fixed, and the test case now works correctly.
- **If fail:** Report that the defect persists, and the fix was unsuccessful.

Output of retesting

- **Pass:** The defect is fixed, and the test case now passes as expected.
- **Fail:** The defect persists or reappears, and the issue has not been resolved as intended.

© NIMI NOT TO BE REPUBLISHED

Key comparison between regression and retesting

Feature	Regression Testing	Retesting
Purpose	To ensure that new changes (bug fixes, features) don't break existing functionality.	To verify that a specific bug or defect has been fixed.
Scope	Broader scope; tests many functionalities across the application.	Narrow scope; focuses only on the specific test case that failed due to the defect.
Test Cases	Test cases are selected based on areas impacted by changes (both past and new failures).	The exact same test case that failed previously is re-executed.
Timing	Performed after changes, updates, or new releases.	Performed after the fix of a specific defect.
Objective	Ensure no new issues have been introduced in the system.	Confirm that a specific defect has been resolved.
Example	Running tests on login, registration, search, etc., after a new "Forgot Password" feature is added.	Running the login test case again to confirm that the bug with password handling is fixed.

Example scenario for both testing types

Scenario: You are testing a website where users log in, register, and reset their passwords.

Regression testing example

i **New feature** A new "Forgot Password" feature is added to the login page.

ii **Selected test cases**

- Test cases for login functionality.
- Test cases for password reset.
- Test cases for user registration.

iii **Execution:** You run these tests to make sure the "Forgot Password" feature works and that the login functionality, registration process, and password reset areas still work correctly.

iv **Output**

- **Pass:** All the tests for login, registration, and password reset pass, confirming no breakage.
- **Fail:** If the login fails due to session issues caused by the "Forgot Password" feature, report a defect.

Retesting example

i **Bug fix:** The "Forgot Password" feature was not sending reset emails to users. The development team fixes this bug.

ii **Selected test case:** The test case that was executed previously and failed (the "Forgot Password" test case).

iii **Execution:** You rerun the "Forgot Password" test case to check if the email issue has been resolved.

iv **Output**

- **Pass:** If the email is now being sent correctly when the user clicks the reset password link.
- **Fail:** If the email issue persists, the bug is not fixed, and the issue is reported again.

© NIMI NOT TO BE REPUBLISHED

— — — — — —

White box testing

Objective: At the end of this exercise you shall be able to
- **white box testing focuses on internal code structure.**

Requirements

Tools/Equipment/Instruments

• A working PC with static analysis tools	- as reqd.

PROCEDURE

White box testing step-by-step with output

White Box Testing (also known as Clear Box Testing, Glass Box Testing, or Structural Testing) is a type of software testing where the tester has access to the internal structure, design, and code of the application. The tester is required to know the implementation of the code to validate the internal operations of the software.

Step-by-step process for white box testing

1 Requirement analysis

- **Purpose:** Understand the functionality and behaviour of the software to design effective test cases.
- **Actions:** Review the software's requirements and gather documentation on the application's architecture, code, and internal logic.

Example output

- Requirement document analysis provides the understanding that the software is a banking application that allows users to deposit, withdraw, and check balance.

2 Code review

- **Purpose:** Review the source code to identify potential areas that need testing, such as logic paths, loops, conditions, and functions.
- **Actions:** Analyze the code structure for things like:
  - Unused variables or dead code
  - Conditional statements
  - Loops and functions
  - Data flow and control flow

Example output

- The code review identifies a function that handles user authentication and another function that processes transactions (deposit and withdrawal).

3 Test case design

- **Purpose:** Based on the code review, design test cases that will test the various paths, loops, and branches in the code.
- **Actions**
  - **Path coverage:** Design test cases that cover every possible path in the code.
  - **Branch coverage:** Design test cases to execute each branch in conditional statements.
  - **Condition coverage:** Design test cases to test each condition within a decision-making construct.
  - **Loop coverage:** Ensure that loops are tested with 0, 1, and multiple iterations.

Example output

- Test cases are created to
  - Test the login function with valid and invalid credentials (covering different branches in the authentication process).
  - Test the withdrawal function for positive and negative amounts (testing loop and condition coverage).

4 Test execution

- **Purpose:** Execute the designed test cases on the application, using the knowledge of the internal code to validate whether the implementation works as expected.
- **Actions:** Run the test cases on the software and monitor the internal operations, like data processing, memory usage, etc.

Example output

- **Test case 1 (Login function):** Test with valid credentials.
  - **Expected:** The user should be logged in successfully.
  - **Result:** Passed – user is logged in.
- **Test case 2 (Withdrawal function):** Test with invalid amount (negative amount).
  - **Expected:** The system should reject the transaction and show an error.
  - **Result:** Failed – No error message was shown, indicating a flaw in the code.

© NIMI NOT TO BE REPUBLISHED

5 Debugging

- **Purpose:** If the test cases fail, debug the code to find the root cause of the problem.
- **Actions:** Use debugging tools to trace the code and fix any issues, such as incorrect logic, missing conditions, or improper data handling.

Example Output

- The bug in the withdrawal function is traced to a missing validation check for negative amounts.
- Developer fixes the issue, and the code is re-deployed.

6 Retesting

- **Purpose:** Once the identified issues are fixed, rerun the failed test cases to confirm that the issue is resolved.
- **Actions:** Execute the same test case again to verify if the issue is fixed and the system now works as expected.

Example output

- **Retest result:** The withdrawal function now properly handles negative amounts and shows an error message.
- Test passes, confirming the issue is fixed.

7 Test closure

- **Purpose:** Once testing is complete, analyze the testing process and prepare a test summary report.
- **Actions:** Compile the results of test cases, including passed and failed tests, and report any outstanding issues or bugs that need further attention.
- Include test coverage metrics to ensure that the code has been adequately tested.
- Document the test cases executed, results, and any code changes made.

Example output

- **Test summary report**
  - Total test cases executed: 20
  - Total passed: 18
  - Total failed: 2
  - Total defects found: 1 (defect in withdrawal function, now fixed)
  - Test Coverage: 95% (all major paths and conditions covered)
  - Test completed successfully.

Summary of white box testing process

S.No.	Step	Action	Output
1	Requirement Analysis	Analyze the requirements and functionality of the system.	Understanding of how the software is intended to work.
2	Code Review	Review the code structure and internal logic.	Identify key areas of the application for testing.
3	Test Case Design	Design test cases based on code logic, paths, branches, loops, etc.	Test cases for path, branch, condition, and loop coverage.
4	Test Execution	Run the test cases on the application.	Test results for each case (Pass/Fail).
5	Debugging	Debug the failed test cases.	Code fixes for the issues identified during testing.
6	Retesting	Execute the test cases again after fixes.	Confirm the bug fix and ensure that the functionality works as expected.
7	Test Closure	Generate the test summary and closure report.	Report on test completion, defect details, and coverage metrics.

© NIMI NOT TO BE REPUBLISHED

Output of white box testing

- **Pass:** The functionality behaves as expected, all paths are executed correctly, and no defects are found.
- **Fail:** If the code fails to behave as expected (e.g., incorrect logic, missing validation), defects are reported.
- **Defects identified:** Any discrepancies or bugs in the code are highlighted and need fixing.

Example of output from test execution

Test case 1: Login function (Valid credentials)

- **Input:** Username: "user", Password: "password123"
- **Expected Outcome:** User successfully logged in.
- **Actual Outcome:** User successfully logged in.
- **Result:** Pass

Test case 2: Withdrawal function (Negative amount)

- **Input:** Amount: -100
- **Expected outcome:** Error message should appear (invalid amount).
- **Actual outcome:** No error message, transaction was processed.
- **Result:** Fail (Bug identified)

Black box testing

Objective: At the end of this exercise you shall be able to
- **black box testing evaluates functionality without code knowledge.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with test management tools	- as reqd.

PROCEDURE

Black box testing step-by-step with output

Black Box Testing is a type of software testing where the tester focuses on testing the functionality of the software without knowing anything about the internal structure or implementation details of the application. The tester only cares about the inputs and the expected outputs.

Step-by-step process for black box testing

1 Requirement analysis

- **Purpose:** Understand the functional requirements of the application to design effective test cases.
- **Actions**
  - Review the software requirements, user stories, and specifications to understand what the software is supposed to do.
  - Focus on what the system should do, not how it does it.

Example output

- The application is a banking system that should allow users to perform transactions like deposit, withdrawal, balance checking, etc.

2 Test case design

- **Purpose:** Design test cases based on the application's expected behaviour, focusing on inputs, outputs, and functional requirements.
- **Actions**
  - Create positive test cases (test valid inputs and expected behaviour).
  - Create negative test cases (test invalid inputs or edge cases).
  - Test the system's behaviour when provided with a variety of inputs and conditions.
  - Utilize techniques like

i **Equivalence partitioning:** Divide inputs into valid and invalid groups and test one representative value from each group.

ii **Boundary value analysis:** Test the boundaries of input values, as they are more likely to cause errors.

iii **Decision table testing:** Test combinations of inputs and their corresponding outputs.

iv **State transition testing:** Test transitions between different states of the application.

Example output

- **Test case 1:** Check if the login page works with valid credentials (positive test).
- **Test case 2:** Test login with invalid credentials (negative test).
- **Test case 3:** Test withdrawal with valid amount (positive test).
- **Test case 4:** Test withdrawal with insufficient balance (negative test).

3 Test execution

- **Purpose:** Execute the designed test cases on the software to validate its functionality.
- **Actions**
  - Run each test case one by one on the application, without any knowledge of the underlying code.
  - Provide inputs (user actions, data entry) and observe the outputs.
  - Test the application's user interface (UI) and check if it responds correctly to each input.

Example output

- **Test case 1 (Login with valid credentials)**
  - **Input:** Username: "user", Password: "password123"
  - **Expected output:** Successful login and redirection to the home page.
  - **Result:** Pass (User is successfully logged in).

© NIMI NOT TO BE REPUBLISHED

- **Test case 2 (Login with invalid credentials)**
  - **Input:** Username: "user", Password: "wrongpassword"
  - **Expected output:** Error message stating "Invalid username or password."
  - **Result:** Pass (Error message displayed as expected).
- **Test case 3 (Withdrawal with sufficient balance)**
  - **Input:** Amount: $500, Account balance: $1000
  - **Expected output:** Successful withdrawal, new balance: $500.
  - **Result:** Pass (Withdrawal successful, balance updated).
- **Test case 4 (Withdrawal with insufficient balance)**
  - **Input:** Amount: $2000, Account balance: $1000
  - **Expected output:** Error message "Insufficient funds."
  - **Result:** Pass (Error message displayed as expected).

4 Defect reporting

- **Purpose:** If a defect is found during the execution, it needs to be reported.
- **Actions:** Document the defects and report them to the development team.
  - Include information such as the test case ID, steps to reproduce, expected behaviour, actual behaviour, and any relevant logs/ screenshots.
  - If no defects are found, proceed to the next phase.

Example output

- **Test case 5 (Balance check with negative balance)**
  - **Input:** Amount: -500, Account balance: $1000
  - **Expected output:** Error message stating "Amount cannot be negative."
  - **Result:** Fail (No error message displayed).
  - **Defect:** The system does not handle negative values properly in the balance check functionality. Reported as defect #1234.

5 Retesting

- **Purpose:** Once the development team fixes the defects, retest the application to verify the fix.
- **Actions:** Execute the same test cases that failed previously to ensure that the defect has been resolved.
  - Ensure that the system now behaves as expected for the particular scenario.

Example output

- After fixing the defect in test case 5
- **Test case 5 (Balance check with negative balance)**
  - i **Input:** Amount: -500, Account balance: $1000
  - ii **Expected output:** Error message stating "Amount cannot be negative."
  - iii **Result:** Pass (Error message displayed as expected).

6 Test closure

- **Purpose:** Conclude the testing process, document findings, and generate a final test report.
- **Actions**
  - Collect and analyze all the executed test cases, report the pass/fail outcomes.
  - Summarize the defects found and their resolution status.
  - Calculate test coverage and determine if any critical areas were missed.
  - Prepare and share the test summary report with stakeholders.

Example output

- **Test summary report**
  - **Total test cases executed:** 20
  - **Total passed:** 18
  - **Total failed**: 2 (Defects reported and fixed).
  - **Test coverage:** 90% (Functional requirements covered).
  - **Defects found:** 2 (Login validation error and negative balance handling issue).
  - **Test status:** Completed successfully.

Summary of black box testing process

S.No	Step	Action	Output
1	Requirement Analysis	Understand the requirements of the system.	Functional specification and user expectations identified.
2	Test Case Design	Design test cases based on inputs, outputs, and functionality.	Test cases based on functional requirements (positive and negative tests).
3	Test Execution	Execute the test cases and validate outputs.	Test results for each case (Pass/Fail).

© NIMI NOT TO BE REPUBLISHED

4	Defect Reporting	Report any defects found during testing.	Defects identified with detailed reports for developers.
5	Retesting	Re-execute failed test cases after defect fixes.	Confirm defect resolution and validate the fix.
6	Test Closure	Generate the final test report.	Test summary with pass/fail statistics, defects, and coverage.

Output of black box testing

- **Pass:** The functionality behaves as expected for all test cases, with no issues found.
- **Fail:** If the system fails to meet the expected outcomes, defects are identified and reported.
- **Defects:** If the application fails to handle invalid inputs or edge cases, defects are raised and sent for fixing.

Example of test results output from test execution

Test case 1: Login function (Valid credentials)

- **Input:** Username: "user", Password: "password123"
- **Expected Outcome:** Successful login and access to the home page.
- **Actual Outcome:** User successfully logged in.
- **Result:** Pass

Test case 2: Withdrawal (Insufficient funds)

- **Input:** Withdrawal amount: $2000, Balance: $1000
- **Expected Outcome:** Error message "Insufficient funds."
- **Actual outcome:** Error message displayed as expected.
- **Result:** Pass

Test case 3: Balance check (Negative amount)

- **Input:** Balance: -500
- **Expected outcome:** Error message "Amount cannot be negative."
- **Actual outcome:** No error message displayed.
- **Result:** Fail (Defect raised).

© NIMI
NOT TO BE REPUBLISHED

White box v/s Black box

Objective: At the end of this exercise you shall be able to
- **white box tests code; black box tests functionality.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with test management tools	- as reqd.

PROCEDURE

White box testing vs black box testing: Step-by-step with output

White Box Testing and Black Box Testing are two fundamental testing approaches that differ in their focus, scope, and techniques. Here's a detailed comparison, step by step, with examples and outputs.

1 Definition

- **White box testing:** Also known as Structural Testing or Clear Box Testing, it tests the internal structures or workings of an application. The tester has access to the internal code and logic of the system.
- **Black box testing:** In this testing, the tester is not concerned with the internal workings of the application. The focus is solely on inputs and expected outputs without knowledge of the internal code.

Comparison

Aspect	White Box Testing	Black Box Testing
Definition	Focuses on internal structure and logic of the software.	Focuses on external functionality and behavior of the software.
Tester Knowledge	Requires knowledge of the code and internal structure.	Tester doesn't need knowledge of the internal code or design.
Focus	Code, control flow, data flow, branches, loops.	Functionality, inputs, outputs, user experience.
Test Case Design	Based on code structure, paths, branches, loops.	Based on requirements, use cases, and user stories.
Techniques	Path testing, branch testing, condition testing.	Equivalence partitioning, boundary value analysis, decision table testing.

2 Test objective

- **White box testing:** To ensure that the code functions correctly, checking the logic, pathways, and internal structures.
- **Black box testing:** To ensure that the software works as intended from the user's perspective and that all functional requirements are met.

Example

- **White box testing:** Test the logic of an addition function
  - **Function:** add(a, b) -> Returns sum of two numbers.
  - **Test case:** Verify if add(1, 1) returns 2 (testing code path).
- **Black box testing:** Test the login functionality
  - **Test case:** Verify if the system correctly logs in with valid credentials (username="user", password="password123").

3 Test case design

- **White box testing:** Test cases are based on the internal code structure.

- **Example techniques**

i **Statement coverage:** Ensuring every line of code is executed at least once.

ii **Branch coverage:** Ensuring all branches (if/else statements) are tested.

iii **Path coverage:** Ensuring all possible paths through the code are tested.

© NIMI NOT TO BE REPUBLISHED

- **Black box testing:** Test cases are based on functional requirements.

- **Example techniques**

  i **Equivalence partitioning:** Dividing input data into valid and invalid partitions.

  ii **Boundary value analysis:** Testing the boundaries of input values.

  iii **Decision table testing:** Testing combinations of inputs and expected outputs.

4 Test execution

White box testing execution

- **Test Case 1:** Test the addition function
  - **Input:** add(2, 3)
  - **Expected output:** 5
  - **Code coverage:** Ensures that the code path for the addition function is fully covered.

Result: Pass (if the output is correct and the code is executed correctly).

- **Test case 2:** Test an if-else condition
  - **Input:** x = 10, check if if (x > 5):
  - **Expected output:** Enter the "true" branch and print "Greater".
  - **Actual output:** "Greater"

Result: Pass (The logic is correct).

Black box testing execution

- **Test case 1:** Test login functionality (valid credentials):
  - **Input:** username="user123", password="password123"
  - **Expected output:** Login successful, user redirected to the dashboard.

Result: Pass (Login successful, as expected).

- **Test case 2:** Test login functionality (invalid password):
  - **Input:** username="user123", password="wrongpassword"
  - **Expected output:** Error message "Invalid credentials."

Result: Pass (Error message shown).

5 Defect detection

- **White box testing:** Helps find defects related to code structure, logic errors, unhandled conditions, and performance issues.
- **Black box testing:** Helps find defects related to functional errors, missing functionality, and usability problems.

Example

- **White box testing defect**
  - **Bug:** A loop in the code does not terminate properly under certain conditions.
  - **Test case:** Check if the loop executes as expected.
  - **Result:** Fail (loop runs indefinitely).
- **Black box testing defect**
  - **Bug:** The login page accepts invalid inputs (e.g., empty fields).
  - **Test case:** Test login with blank username or password.
  - **Result:** Fail (blank credentials are accepted).

6 Test coverage

- **White box testing:** Provides high code coverage, ensuring that all paths, conditions, and branches are tested.
- **Black box testing:** Provides high functional coverage but doesn't guarantee full code coverage.

7 Advantages

- **White box testing**
  - Provides complete code coverage.
  - Helps identify hidden errors, such as logic errors and untested code paths.
  - Ensures optimization of the code.
- **Black box testing**
  - Focuses on user functionality and real-world usage.
  - Requires no knowledge of the internal code, which makes it easier to perform from a user perspective.
  - Helps test non-functional aspects like UI/UX, performance, and security.

8 Disadvantages

- **White box testing**
  - Requires access to the code and programming skills.
  - Can miss unanticipated issues that arise during real-world use.
- **Black box testing**
  - Does not guarantee full code coverage.
  - Harder to identify root causes of failures (since no access to code).

© NIMI NOT TO BE REPUBLISHED

9 Example: A simple addition function

White box testing

Let's assume we have the following addition function in code. (Fig 1)

Fig 1

```
def add(a, b):
    if a == 0 or b == 0:
        return 0
    else:
        return a + b
```

Test case 1: Check the behaviour when one input is zero.

- **Input:** add(0, 5)
- **Expected output:** 0
- **Code path:** The function enters the if branch (a == 0).

Result: Pass

Test case 2: Check the behaviour when both inputs are non-zero.

- **Input:** add(3, 5)
- **Expected output:** 8
- **Code path:** The function enters the else branch.

Result: Pass

Black box testing

We have a login functionality where a user must provide a valid username and password.

- **Test case 1:** Test login with valid credentials.
  - **Input:** username="user123", password="password123"
  - **Expected output:** Login successful, user redirected to the dashboard.

Result: Pass

- **Test case 2:** Test login with invalid credentials.
  - **Input:** username="user123", password="wrongpassword"
  - **Expected output:** Error message "Invalid credentials."

Result: Pass

10 Summary Comparison of White Box vs Black Box Testing

Aspect	White Box Testing	Black Box Testing
Focus	Internal logic and code structure.	Functional behavior and user experience.
Test Case Design	Based on code paths, branches, and conditions.	Based on functional specifications and requirements.
Knowledge Required	Knowledge of internal code and logic.	No knowledge of internal code required.
Defect Detection	Finds issues in code structure and logic.	Finds issues in functionality and user interface.
Test Coverage	Provides high code coverage.	Provides high functional coverage.
Example	Testing if a loop terminates correctly.	Testing if the login page works with valid and invalid credentials.

© NIMI NOT TO BE REPUBLISHED

Verification & Validation

Objective: At the end of this exercise you shall be able to
- **verification ensures correctness; validation ensures requirements fulfilment.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with test management tools	- as reqd.

PROCEDURE

Verification and Validation are two important processes in software development and systems engineering, ensuring that the product meets its requirements and works as expected. Let's break them down step by step

1 Verification : Verification ensures that the product is built correctly. It checks whether the product meets the specifications and design requirements. Verification is often referred to as "building the product right."

Step-by-step process for verification

i Requirement review

- **Output:** Detailed understanding of functional and non-functional requirements.
- Ensure that the requirements are clear, complete, consistent, and traceable.

ii Design review

- **Output:** Design documents or models that describe how the requirements will be met.
- Ensure that the design accurately reflects the requirements and is feasible.

iii Code review

- **Output:** Verified source code.
- Check if the code follows coding standards, guidelines, and meets the design specifications.

iv Unit testing

- **Output:** Unit test results (pass/fail for individual code units).
- Each individual unit/module is tested for correctness, ensuring it works as intended.

v Integration testing

- **Output:** Results showing whether the integrated modules work together.
- After units are tested individually, integration tests check that components work together.

vi Static analysis

- **Output:** A report on potential code issues (e.g., memory leaks, unused variables).
- Automated tools analyze the source code for potential problems without executing it.

vii System testing

- **Output:** Test results showing that the system as a whole functions according to the design.
- The whole system is tested in an environment that simulates real-world conditions.

viii Traceability matrix

- **Output:** A matrix that traces each requirement to corresponding test cases.
- Ensures that every requirement has been tested and validated.

2 Validation : Validation ensures that the product fulfills the user's needs and performs the intended tasks in real-world scenarios. It's about ensuring the product is "the right product."

Step-by-step process for validation

i Requirement validation

- **Output:** Validated requirements.
- Ensure that the initial requirements accurately reflect user needs and business goals.

ii Prototyping or Simulation

- **Output:** A prototype or simulation of the product.
- Develop a working model (could be a prototype) to gather early feedback from users or stakeholders.

iii User Acceptance Testing (UAT)

- **Output:** UAT feedback (user acceptance or rejection).

© NIMI NOT TO BE REPUBLISHED

- Test with real users or clients to validate whether the system meets their expectations and needs.

iv Alpha testing

- **Output:** Internal testing feedback from developers and testers.
- Conduct an initial round of testing within the development team to identify bugs and issues.

v Beta testing

- **Output:** Feedback from a selected group of users.
- Test the product with real users in a controlled, limited release, gathering feedback and fixing issues.

vi Performance testing

- **Output:** Test results for performance under load.
- Ensure the system performs well under expected and peak conditions (e.g., load, stress, and scalability testing).

vii Security testing

- **Output:** Security vulnerability report.
- Validate that the system is secure and meets security requirements, including data protection and authentication.

viiiRegulatory compliance testing

- **Output:** Compliance status report.
- Ensure the system adheres to regulatory standards and laws (e.g., GDPR, HIPAA).

Example output for verification and validation

- **Verification output example**
  - Code review report: No critical errors found, minor style improvements suggested.
  - Unit testing report: 95% test coverage, 2 tests failed due to edge case handling.
  - Integration testing result: All modules integrate successfully, no data loss during transfer.
  - System testing result: All system features work according to specifications, with minor performance issues on large datasets.
- **Validation output example**
  - UAT feedback: 90% of users report satisfaction, but 10% request more intuitive navigation.
  - Beta testing feedback: Identified a bug in the payment gateway, users report great performance on mobile devices.
  - Performance testing: System handles 10,000 concurrent users with a response time under 2 seconds.
  - Security testing: No critical vulnerabilities found; minor issues with weak password policies.

© NIMI NOT TO BE REPUBLISHED

Acceptance Testing

Objective: At the end of this exercise you shall be able to
- **acceptance testing confirms software meets user requirements.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with test management tools	- as reqd.

PROCEDURE

Acceptance testing is a critical phase in software development, conducted to ensure the software meets the business requirements and is ready for deployment. It is typically the final step before the product goes live.

Here's a step-by-step guide to performing acceptance testing

Step 1: Prepare the test plan

- **Define scope and objectives:** Clearly state what needs to be tested, including features, functionalities, and any areas of the system that need to be verified against the requirements.
- **Determine acceptance criteria:** These criteria are the conditions the product must meet in order to be accepted.
- **Select acceptance testers:** Involve stakeholders, such as end users, business analysts, and other relevant parties, to carry out testing.

Output: Test plan document including test objectives, scope, acceptance criteria, and selected testers.

Step 2: Test environment setup

- **Identify the test environment:** Ensure the testing environment mimics production as closely as possible.
- **Set up test data:** Prepare test data that mirrors real-life scenarios, including valid and invalid inputs.

Output: A fully configured test environment, including necessary hardware, software, network configurations, and test data.

Step 3: Design test cases

- **Map business requirements to test cases:** For each feature, design test cases that cover normal use cases, edge cases, and failure scenarios.
- **Write clear and concise test scripts:** Ensure test cases are easy to understand and executable. Include expected results.

Output: A collection of test cases that reflect the requirements and define the expected results.

Step 4: Execute acceptance tests

- **Run the test cases:** Execute each test case in the test environment, ensuring the software behaves as expected according to the acceptance criteria.
- **Document test results:** For each test, record whether it passed or failed, and any issues encountered during execution.

Output: Test execution results indicating pass/fail status for each test case.

Step 5: Report defects and issues

- **Log defects:** If any test case fails, log the defect with detailed steps to reproduce, the expected behaviour, and the actual outcome.
- **Assign priorities:** Depending on the severity of the defects, assign priorities (e.g., critical, high, medium, low) for fixing.

Output: Defect report, including a list of all issues found with priorities and status.

Step 6: Review and fix issues

- **Development team resolves defects:** Developers work on fixing the issues found in the testing phase.
- **Re-test:** Once defects are fixed, re-test the impacted areas to ensure the problems are resolved.

Output: Updated system with defect fixes and re-test results showing the fixes work.

Step 7: Sign-off

- **Acceptance decision:** Stakeholders review the test results and the fixed issues. If the product meets the acceptance criteria, they approve the product for release.
- **Final report:** Document the test results, including pass/fail status, defects, and overall decision.

Output: Signed-off acceptance report, confirming the software is ready for release.

© NIMI NOT TO BE REPUBLISHED

Example output scenario

1 **Test plan document**

- **Objectives:** Verify the functionality of the user registration feature.
- **Criteria:** Correct data validation, successful account creation, email confirmation.
- **Testers:** Product owner, end users, QA team.

2 **Test execution results**

- **Test case 1:** Valid email input → Passed.
- **Test case 2:** Invalid email input → Failed (error message not shown correctly).

3 **Defect report**

- **Defect ID:** 001
- **Description:** "Email validation error message is not showing for invalid input."
- **Severity:** High
- **Status:** Open

4 **Final sign-off report**

- **Overall Status:** Passed with one high-priority defect logged.
- **Recommendation:** Fix defect and re-test before the final deployment.

© NIMI
NOT TO BE REPUBLISHED

Non-Functional Testing

Objective: At the end of this exercise you shall be able to
- **non-functional testing evaluates performance, usability, and security.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with test management tools	- as reqd.

PROCEDURE

Non-functional testing focuses on how well the software performs, rather than what it does. It checks the non-functional aspects of the application such as performance, scalability, security, and usability.

Here's a step-by-step guide to performing non-functional testing

Step 1: Prepare the non-functional test plan

- **Identify non-functional requirements:** Review the project's non-functional requirements like performance, load, security, etc.
- **Determine test objectives:** Specify what you want to test (e.g., system performance, stress, usability, security, etc.).
- **Choose the tools:** Decide which tools will be used (e.g., JMeter for load testing, Selenium for usability testing, etc.).
- **Define the test environment:** Prepare the hardware and software setup for testing, similar to the production environment.

Output: Non-functional test plan document, which includes the objectives, testing tools, environment setup, and test schedule.

Step 2: Set up the test environment

- **Set up hardware and software:** Ensure the environment mimics the production system. This could include servers, network configurations, and configurations that affect performance.
- **Install testing tools:** Set up the necessary tools for performance testing (like JMeter or LoadRunner), security testing (like OWASP ZAP), and other non-functional tests.
- **Prepare test data:** Create test data based on the requirements. This includes large datasets for load testing or realistic data for usability tests.

Output: A test environment with all necessary configurations, tools, and data for non-functional testing.

Step 3: Design non-functional test cases

- **Performance testing:** Design tests that simulate user load, stress, and scalability. Identify key performance indicators (KPIs) such as response time, throughput, and resource utilization.
  - **Example test case:** "Simulate 1000 users logging in simultaneously. Measure the response time and resource usage."
- **Security testing:** Design tests to identify vulnerabilities. This may include testing for SQL injection, cross-site scripting (XSS), authentication weaknesses, etc.
  - **Example test case:** "Test for SQL injection by submitting SQL code in the input fields."
- **Usability testing:** Design tests to evaluate the user-friendliness and accessibility of the application.
  - **Example test case:** "Evaluate the ease of navigating the user interface for first-time users."
- **Reliability and availability testing:** Test how well the system behaves under continuous use and how it recovers from failures.
  - **Example test case:** "Simulate a server crash and test how the system recovers after restart."

Output: A collection of detailed non-functional test cases for performance, security, usability, and reliability.

Step 4: Execute non-functional tests

- **Run performance tests:** Use tools like JMeter or LoadRunner to simulate users accessing the system under various conditions (e.g., normal load, peak load).
  - For example, simulate 100, 500, and 1000 users logging in and track response time, errors, and resource usage.
- **Conduct security testing:** Use tools such as OWASP ZAP to find vulnerabilities like SQL injections, XSS, and insecure API endpoints.
- **Perform usability testing:** Have end users test the system to identify navigation issues, UI design flaws, or accessibility problems.
- **Test reliability:** Simulate system failures or heavy loads to see how the system behaves over time (e.g., how it recovers after downtime).

© NIMI NOT TO BE REPUBLISHED

Output: Results of test executions, which include performance metrics (e.g., response times, error rates), security vulnerabilities, usability feedback, and reliability results.

Step 5: Analyze test results

- **Analyze performance results:** Compare the actual performance against the expected benchmarks. Check if the system meets the required performance under stress.
  - **Example:** "Response time under 1000 users exceeded the acceptable threshold of 2 seconds."
- **Analyze security results:** Identify any vulnerabilities discovered during security testing.
  - **Example:** "Found a potential SQL injection vulnerability in the login form."
- **Analyze usability feedback:** Compile the user feedback and look for patterns that indicate areas of improvement in the UI/UX.
  - **Example:** "Users struggled to find the 'Submit' button, suggesting a poor UI layout."
- **Analyze reliability results:** Determine the system's behaviour during crashes or failures. Check if it recovers properly and within the expected timeframe.
  - **Example:** "System failed to recover after server crash and required manual intervention."

Output: Test analysis report, summarizing performance, security, usability, and reliability results, along with areas for improvement.

Step 6: Report issues and defects

- **Log performance bottlenecks:** Record any performance issues such as high response times, resource exhaustion, or errors under load.
  - **Example:** "Page load time for 1000 users is 5 seconds, which is above the expected threshold of 2 seconds."
- **Log security defects:** Record discovered vulnerabilities along with steps to reproduce and impact analysis.
  - **Example:** "SQL injection vulnerability discovered in the login form. Affected areas include user authentication."
- **Log usability issues:** Identify UI/UX issues that reduce user experience.
  - **Example:** "Users experienced difficulty in navigating between pages, especially on mobile devices."
- **Log reliability defects:** Document issues related to system recovery and failure rates.
  - **Example:** "System crashed and required manual intervention for recovery after 48 hours of continuous use."

Output: Defect report listing performance issues, security vulnerabilities, usability problems, and reliability concerns.

Step 7: Fix and re-test issues

- **Address performance issues:** Optimize the system by identifying the bottleneck and improving resource utilization.
- **Resolve security vulnerabilities:** Fix the security issues (e.g., apply patches, fix insecure code, implement better encryption).
- **Improve usability:** Implement design changes based on usability feedback.
- **Improve reliability:** Enhance fault tolerance and recovery mechanisms.

Output: Updated system with fixes for performance, security, usability, and reliability issues.

Step 8: Sign-off

- **Stakeholder review:** After fixing issues, stakeholders review the updated results. If the product meets all non-functional requirements, they approve it for release.
- **Final report:** Document all non-functional testing results, including performance, security, usability, and reliability assessments.

Output: Final non-functional testing report, summarizing all test results, issues found, fixes applied, and overall system performance.

© NIMI NOT TO BE REPUBLISHED

Usability Testing

Objective: At the end of this exercise you shall be able to
- **usability testing ensures software is user-friendly and intuitive.**

Requirements

Tools/Equipment/Instruments

- A working PC with test management tools - as reqd.

PROCEDURE

Usability testing is a type of non-functional testing that focuses on assessing how user-friendly and intuitive a product is. It involves evaluating how easily users can interact with a system or application to achieve their goals. Here's a step-by-step guide to conducting usability testing, along with example outputs

Step 1: Define objectives and goals

- **Identify the purpose of the usability test:** Determine which specific aspects of the user experience you want to evaluate (e.g., navigation, interface layout, content comprehension, etc.).
- **Set measurable goals:** Goals should focus on key usability metrics such as efficiency, effectiveness, and satisfaction.
  - **Example goal:** "Test whether users can complete a purchase transaction within 5 minutes."

Output: Usability test plan that includes objectives like "Evaluate the ease of navigating the checkout process" and measurable goals like "Ensure users complete the checkout in under 5 minutes."

Step 2: Define the test scope

- **Select features for testing:** Focus on the critical features that users will interact with the most (e.g., login, product search, shopping cart, or payment process).
- **Target audience:** Define the user group who will participate in the test. Choose participants who represent the actual users of the system.
  - **Example:** "Targeting new users who have never interacted with the application before."

Output: A list of features to be tested, such as the product search feature, and a profile of the target audience (e.g., "New users between ages 18-40 who are familiar with online shopping but new to our website").

Step 3: Create usability test scenarios

- **Develop realistic tasks:** Create tasks that simulate real-world scenarios users will encounter while using the application.
  - **Example tasks:** "Search for a red dress", "Add a product to the cart", "Complete the checkout process."
- **Define success criteria:** For each task, define what constitutes success (e.g., the user can complete the task without assistance).

Output: A list of test scenarios and success criteria, for example

- **Task 1:** Search for a red dress → Success if the user finds the red dress within 2 minutes.
- **Task 2:** Add a product to the cart → Success if the user successfully adds a product within 30 seconds.

Step 4: Recruit test participants

- **Select users:** Choose participants who are representative of the actual target audience. You may recruit 5-10 participants for a comprehensive test.
  - Participants may include both experienced and novice users depending on the goals of the test.
- **Prepare consent forms:** Have participants sign consent forms to ensure they agree to be part of the study.

Output: A pool of recruited participants, including names, demographics, and consent forms signed.

Step 5: Conduct the test

- **Moderated or unmoderated testing:** Decide whether you will have a facilitator guiding the users through the tasks (moderated) or if users will perform the tasks independently (unmoderated).
  - **Moderated testing:** A facilitator is present to observe and take notes. The facilitator may ask questions to gain insight into the user's thinking.
  - **Unmoderated testing:** The user performs the test independently, with pre-recorded instructions.
- **Observe user behaviour:** Pay attention to users' facial expressions, body language, and comments. Also, track how long it takes to complete tasks, if they need help, and where they struggle.

© NIMI NOT TO BE REPUBLISHED

- **Record interactions:** Use screen recording tools or observe in person to capture user actions.

Output: Observational data, including task completion time, user comments, and areas where the user struggles or succeeds. For example:

- **Task 1: Search for a red dress:** Completed in 3 minutes. User comments: "I had trouble finding the search bar."
- **Task 2: Add to cart:** Completed in 10 seconds. No issues encountered.

Step 6: Analyze results

- **Identify usability issues:** Review the recordings and notes to identify patterns in user behaviour. Look for areas where users struggled, made errors, or expressed frustration.
  - **Example:** "Users had difficulty finding the 'Add to Cart' button because it was too small."
- **Quantitative analysis:** Measure the success rate of each task (e.g., number of successful completions), average time taken, and number of errors.
- **Qualitative analysis:** Examine user comments, facial expressions, and behavioural patterns for deeper insights into their experience.

Output: Usability analysis report, highlighting the following

- **Task 1:** Search for a red dress → 5 out of 10 users struggled to find the search bar.
- **Task 2:** Add to cart → 9 out of 10 users completed it successfully, but one user did not notice the button due to its small size.

Step 7: Generate Usability Issues and Recommendations

- **List identified issues:** Compile all usability problems, categorize them based on severity (e.g., minor, major, critical), and provide specific examples from testing.
  - **Example issue:** "The search bar is hidden at the top of the page, which confuses users."
- **Provide recommendations:** Suggest changes to improve usability, such as UI redesigns or better placement of features.
  - **Example recommendation:** "Make the search bar more prominent by increasing its size and moving it to the center of the page."

Output: A report of usability issues with associated recommendations

- **Issue 1:** Search bar placement → Critical: Move it to a more visible location.
- **Issue 2:** Add to cart button → Minor: Increase the button size for better visibility.

Step 8: Make necessary changes

- **Implement improvements:** The development team makes changes based on the findings from the usability test, such as redesigning UI elements, fixing navigation issues, or optimizing the user flow.
- **Retest if needed:** If significant changes were made, conduct another round of usability testing to verify that the issues were fixed and the user experience has improved.

Output: Updated application with improved usability based on feedback.

Step 9: Final report and sign-off

- **Document findings:** Create a final usability testing report summarizing the process, test scenarios, key findings, issues identified, and changes made.
- **Stakeholder review:** Share the report with stakeholders to get approval for the changes or additional rounds of testing if necessary.

Output: A final usability testing report with

- Overview of the test process
- Key findings (usability issues)
- Improvements made and their impact
- Any further recommendations for optimization.

Example output scenario

1 **Test scenario: Search for a red dress**
   - **Task completion:** 5 out of 10 users found it within 2 minutes.
   - **Usability issue:** "Users struggled to locate the search bar; it was too small and located at the top of the page."
   - **Recommendation:** "Increase the size of the search bar and move it to the center of the page for better visibility."

2 **Test scenario: Add an item to the cart**
   - **Task completion:** 9 out of 10 users succeeded in under 30 seconds.
   - **Usability issue:** "One user missed the 'Add to Cart' button because it was too small."
   - **Recommendation:** "Increase the size of the 'Add to Cart' button for better visibility."

3 **Final report summary**
   - **Usability issues:** Two major issues identified: search bar placement and button visibility.
   - **Improvements:** Search bar was relocated and resized, and the 'Add to Cart' button was made more prominent.
   - **Outcome:** The product is now more user-friendly, with a reduction in errors and faster task completion times.

© NIMI NOT TO BE REPUBLISHED

Stress testing

Objective: At the end of this exercise you shall be able to
- **stress testing evaluates system performance under extreme conditions.**

Requirements

Tools/Equipment/Instruments

• A working PC with test management tools	- as reqd.

PROCEDURE

Stress testing is a type of performance testing that evaluates how a system performs under extreme conditions. The goal is to determine the system's breaking point and how it handles high loads, unexpected spikes, and stress scenarios. Here's a step-by-step guide to performing stress testing, along with example outputs

Step 1: Define stress testing objectives

- **Identify stress scenarios:** Determine what specific stress scenarios need to be tested, such as peak loads, maximum number of users, or extreme data input.
  - **Example:** "Test how the system handles 10,000 concurrent users accessing the website."
- **Set performance benchmarks:** Define acceptable thresholds for system performance, such as response time, CPU usage, and memory utilization.
  - **Example:** "Response time should not exceed 5 seconds under normal load and 10 seconds under stress."

Output: Stress testing objectives document that includes

- **Scenario:** "Test the system under 10,000 concurrent users."
- **Benchmarks:** "System response time must remain under 10 seconds."

Step 2: Design stress test scenarios

- **Create realistic stress scenarios:** Develop scenarios that simulate high load situations, such as a sudden surge in traffic, heavy data uploads, or simultaneous transactions.
  - **Example scenario:** "Simulate 10,000 users simultaneously adding items to their cart and completing purchases."
- **Set up test data:** Prepare the test environment with realistic test data, such as large user profiles or transaction records.
  - **Example:** "Create 10,000 test accounts and 1,000 items in the product catalog."

Output: A list of stress test scenarios, such as

- **Scenario 1:** 10,000 concurrent users adding products to their cart.
- **Scenario 2:** 1,000 simultaneous transactions on the checkout page.

Step 3: Set up the stress testing environment

- **Test environment setup:** Prepare a testing environment that mirrors the production setup. This should include the necessary hardware, software, databases, and network configurations.
- **Choose stress testing tools:** Select appropriate tools to simulate heavy loads and monitor the system performance (e.g., Apache JMeter, LoadRunner, or Gatling).
  - **Example:** "Using Apache JMeter to simulate 10,000 concurrent users."

Output: A fully configured test environment, including

- **Tools:** Apache JMeter for load simulation.
- **Test data:** 10,000 test users and a populated product catalog.

Step 4: Execute the stress tests

- **Run stress tests:** Execute the test scenarios, gradually increasing the load to simulate heavy user traffic or high system load.
  - **Example:** Start with 1,000 users, and gradually increase the number until 10,000 users are simulated.
- **Monitor system performance:** Track system behaviour during the test, including response times, error rates, CPU and memory usage, and database performance.
  - **Example:** Monitor response time, system crash occurrences, or slowdowns under stress.

Output: Test execution data, including

- **Load level 1 (1,000 users):** System runs smoothly with response times under 3 seconds.
- **Load level 2 (5,000 users):** Some minor slowdowns, response time averages 4 seconds.
- **Load level 3 (10,000 users):** System crashes or response time exceeds 10 seconds.

Step 5: Analyze stress test results

- **Identify system limits:** Evaluate at what point the system starts to show performance degradation or failure.

© NIMI NOT TO BE REPUBLISHED

- **Example:** "The system began to fail after 8,000 users, and response time exceeded 12 seconds."

- **Examine resource utilization:** Monitor system resources (CPU, memory, disk I/O) to identify potential bottlenecks.
  - **Example:** "CPU usage peaked at 95% when simulating 8,000 users."

Output: Stress test analysis report

- System failure occurred after 8,000 concurrent users.
- Resource utilization peaked at 95% CPU usage during stress tests.

Step 6: Document Issues and Defects

- **Log failures:** Record any performance issues or system crashes that occurred during the test.
  - **Example:** "System crashes occurred after 8,500 users due to database connection overload."
- **Analyze root cause:** Investigate the cause of failures, such as memory leaks, server limitations, or database bottlenecks.
  - **Example:** "Database connection pooling was exhausted, causing system crashes after a certain load."

Output: Defect report, including

- **Issue 1:** System crashes after 8,500 users due to exhausted database connections.
- **Issue 2:** CPU usage exceeded acceptable levels during peak load.

Step 7: Provide recommendations and optimizations

- **Identify areas for improvement:** Based on the stress testing results, provide recommendations for performance optimization.
  - **Example**: "Increase database connection pool size to handle more simultaneous connections."
- **Suggest infrastructure upgrades:** Recommend upgrades such as scaling the server infrastructure, improving caching, or optimizing database queries.
  - **Example:** "Implement horizontal scaling to handle more users and prevent crashes."

Output: Recommendations report, including

- **Recommendation 1:** Increase database connection pool size to handle up to 10,000 concurrent users.
- **Recommendation 2:** Implement load balancing and server scaling to distribute traffic.

Step 8: Fix issues and re-test

- **Resolve identified issues:** Work with the development and infrastructure teams to fix the issues found during the stress test.
  - **Example:** "Increase the maximum number of database connections to 500."
- **Retest:** Run stress tests again after the issues have been addressed to verify that the fixes were effective and the system can handle the intended load.
  - **Example:** "Re-test with 10,000 users to ensure the system remains stable."

Output: Updated system with fixes, followed by re-test results

- **Re-test result:** After increasing the database connection pool and implementing load balancing, the system handled 10,000 users without failure.

Step 9: Sign-off and report

- **Final reporting:** Create a comprehensive stress testing report that includes the testing objectives, methodology, results, issues identified, and improvements made.
- **Stakeholder review:** Share the report with stakeholders for final approval before proceeding to production.

Output: Final stress testing report that includes

- Overview of the test process and scenarios.
- Summary of test results, including system behaviour under load.
- List of issues and performance bottlenecks.
- Final recommendations and fixes.

Example output scenario

1 Stress test scenario

- **Scenario:** Simulate 10,000 users adding products to the cart and completing transactions.
- **Test execution results**
  - **1,000 users:** No performance issues; response time = 3 seconds.
  - **5,000 users:** Minor performance degradation; response time = 4 seconds.
  - **8,000 users:** CPU usage peaks at 85%, response time = 7 seconds.
  - **10,000 users:** System crashes due to database connection exhaustion.

2 Issues identified

- **Issue 1:** System crashes after 8,000 users due to database connection pool exhaustion.
- **Issue 2:** High CPU utilization when simulating 10,000 users.

3 Recommendations

- **Recommendation 1:** Increase database connection pool size to 500 to handle more simultaneous users.
- **Recommendation 2:** Implement load balancing across multiple servers to distribute traffic and prevent overloading.

4 Re-test result: Re-test with 10,000 users: After fixing the database connection pool and implementing load balancing, the system handled the load without any failures. Response time remained within acceptable limits (less than 10 seconds).

5 Final report

- The system now supports 10,000 concurrent users with no crashes or performance degradation.
- Recommendations for further scalability (e.g., horizontal scaling) are in place to handle future traffic surges.

© NIMI NOT TO BE REPUBLISHED

Load Testing

Objective: At the end of this exercise you shall be able to
- **load testing assesses system performance under expected load.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with test management tools	- as reqd.

PROCEDURE

Load Testing is a type of performance testing that simulates real-world load conditions on a system to ensure it performs as expected under normal usage conditions. Unlike stress testing, which tests the system's limits, load testing focuses on how the system handles expected load over time.

Here's a step-by-step guide to conducting Load Testing, along with examples of expected outputs

Step 1: Define load testing objectives

- **Identify expected load:** Determine the expected number of users or transactions that the system should handle during peak usage. This helps in defining the test load.
  - **Example:** "The system should support 1,000 concurrent users on the homepage without significant performance degradation."
- **Set performance goals:** Set benchmarks for response times, throughput, and error rates that represent acceptable performance levels.
  - **Example:** "Response time should be under 3 seconds for 95% of user requests."

Output: A document outlining load testing objectives

- **Scenario:** "Simulate 1,000 concurrent users."
- **Benchmarks:** "Response time ≤ 3 seconds for 95% of requests."

Step 2: Identify key scenarios to test

- **Select representative scenarios:** Choose the key business processes or system features to simulate, such as user login, product search, or purchase transactions.
  - **Example:** "Simulate a scenario where users are browsing products, adding them to their cart, and completing the checkout process."
- **Define realistic traffic patterns:** Identify typical user behaviour to simulate, such as browsing for 5 minutes and then checking out.
  - **Example:** "80% of users will browse products, 15% will add to the cart, and 5% will complete a purchase."

Output: Load testing scenarios document

- **Scenario 1:** Browse products for 5 minutes.
- **Scenario 2:** Add products to the cart.
- **Scenario 3:** Complete a purchase transaction.

Step 3: Prepare test environment

- **Replicate production environment:** Set up a testing environment that closely matches the production environment in terms of hardware, software, databases, and network configurations.
- **Test data setup:** Create necessary test data for users, products, and transactions to simulate realistic usage.
  - **Example:** "Create 1,000 test users and populate the product catalog with 500 items."

Output: Test environment set up

- Tools: Apache JMeter or LoadRunner for load simulation.
- Test data: 1,000 users, 500 products.

Step 4: Choose load testing tools

- **Select appropriate tools:** Pick a load testing tool that allows you to simulate the expected number of users and capture system performance data.
  - **Examples:** Apache JMeter, LoadRunner, Gatling, or BlazeMeter.

Output: Load testing tool selection

- **Tool:** Apache JMeter.
- **Purpose:** Simulate up to 1,000 concurrent users on the website.

Step 5: Design load test cases

- **Create load test scripts:** Develop scripts for each scenario identified in Step 2. The scripts simulate user actions, such as logging in, browsing, and adding items to the cart.
  - **Example:** "Create a script for simulating a user logging in, browsing products for 5 minutes, adding an item to the cart, and checking out."

© NIMI NOT TO BE REPUBLISHED

- **Define load distribution:** Specify how the load will be distributed across various actions. For example, 70% of users might perform one task, while 30% perform another.
  - **Example:** "70% of users will browse products, 30% will add to the cart."

Output: Load test cases and distribution

- **Script 1:** Simulate browsing for 5 minutes.
- **Script 2:** Simulate adding products to the cart (30% of users).
- **Script 3:** Simulate completing a purchase (20% of users).

Step 6: Execute the load tests

- **Run the tests:** Execute the test cases with varying levels of load. Start with a baseline (e.g., 100 concurrent users) and gradually increase to the target load (e.g., 1,000 concurrent users).
  - **Example:** Start with 100 users, and increase the load by 100 users every 5 minutes until reaching 1,000 users.
- **Monitor system performance:** Track key performance indicators (KPIs), such as response time, CPU usage, memory consumption, error rates, and throughput.
  - **Example**: "Monitor CPU usage, network latency, and error rates while simulating 1,000 users."

Output: Load test execution data

- **At 100 users:** Response time = 2 seconds, CPU usage = 45%, no errors.
- **At 500 users:** Response time = 2.5 seconds, CPU usage = 70%, no errors.
- **At 1,000 users:** Response time = 3.2 seconds, CPU usage = 85%, 1% errors (timeouts).

Step 7: Analyze load test results

- **Evaluate response time:** Compare actual response times against the defined benchmarks (e.g., ≤ 3 seconds).
  - **Example:** "At 1,000 users, response time is 3.2 seconds, which is slightly above the benchmark."
- **Assess system behaviour under load:** Analyze CPU, memory usage, and error rates. Identify any performance bottlenecks or resource limitations.
  - **Example:** "The CPU usage exceeded 80% at 1,000 users, which may indicate server resource constraints."

Output: Load test analysis report

- **Response time:** Exceeds 3 seconds at 1,000 users.
- **CPU usage:** Peaks at 85%, suggesting potential server overload.
- **Error rate:** 1% errors at 1,000 users (timeouts).

Step 8: Identify performance bottlenecks

- **Investigate issues:** Based on the results, identify where the system failed to meet performance goals. Look for potential bottlenecks in the database, server, network, or application code.
  - **Example:** "The database response time increased significantly at 1,000 users, suggesting that it may be a bottleneck."
- **Check resource utilization:** Examine resource utilization, such as CPU, memory, disk I/O, and network usage, to find performance issues.
  - **Example:** "The server CPU usage reached 85%, indicating possible server overload under heavy load."

Output: Bottleneck report

- **Issue 1:** Database response time increased under load.
- **Issue 2:** CPU usage exceeded 80% at peak load.

Step 9: Make optimizations and retest

- **Fix identified issues:** Work with the development or operations teams to resolve performance issues. This could involve database optimization, code optimization, or adding more server capacity.
  - **Example:** "Optimize database queries and increase server capacity to handle more users."
- **Re-run the load tests:** After making the necessary changes, retest the system with the same load levels to verify that performance has improved.
  - **Example:** "After optimizing the database queries and adding more server resources, re-test with 1,000 users."

Output: Retest results

- **After optimization:** Response time = 2.8 seconds at 1,000 users, CPU usage = 75%, no errors.

Step 10: Generate final load testing report

- **Document results:** Prepare a final report that summarizes the objectives, test scenarios, methodology, results, issues found, optimizations made, and final performance metrics.
- **Example:** "The system successfully handled 1,000 users after database optimizations and server upgrades."
- **Present findings to stakeholders:** Share the report with relevant stakeholders, such as developers, project managers, or clients, for review and approval.

Output: Final load testing report

- **Overview:** System load tested with 1,000 concurrent users.
- **Findings:** Response time exceeded benchmarks at 1,000 users; CPU usage peaked at 85%.

© NIMI NOT TO BE REPUBLISHED

- **Recommendations:** Optimized database queries and scaled server infrastructure to handle higher loads.
- **Outcome:** After optimizations, the system handled 1,000 users with a response time of 2.8 seconds and no errors.

Example load test results

1 **Test scenario:** Simulate 1,000 concurrent users browsing products and completing checkout.
   - **Test execution data**
     - **At 100 users:** Response time = 2 seconds, CPU usage = 45%, no errors.
     - **At 500 users:** Response time = 2.5 seconds, CPU usage = 70%, no errors.
     - **At 1,000 users:** Response time = 3.2 seconds, CPU usage = 85%, 1% error rate (timeouts).

2 **Issues identified**
   - **Issue 1:** Response time exceeded 3 seconds at 1,000 users.
   - **Issue 2:** CPU usage exceeded 80% at 1,000 users.
   - **Issue 3:** Database response time increased significantly under load.

3 **Optimizations made**
   - **Database optimization:** Improved query performance.
   - **Server scaling:** Added more server capacity to handle high traffic.

4 **Retest results**
   - **After optimization:** The system successfully handled 1,000 users with a response time of 2.8 seconds, CPU usage at 75%, and no errors.

© NIMI
NOT TO BE REPUBLISHED

Performance Testing

Objective: At the end of this exercise you shall be able to
- **performance testing evaluates system responsiveness and stability.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with test management tools	- as reqd.

PROCEDURE

Performance Testing is a type of testing that evaluates how a system performs under varying conditions. The main objective is to ensure that the system operates within expected performance standards and can handle anticipated user loads. This type of testing can include different tests like load testing, stress testing, and scalability testing.

Here's a step-by-step guide for performing Performance Testing, including expected outputs at each stage

Step 1: Define performance testing objectives

- **Identify key performance metrics:** Clearly define the key performance indicators (KPIs) to measure, such as response time, throughput, resource utilization (CPU, memory), and error rates.
  - **Example:** "The system should handle 1,000 users simultaneously without the response time exceeding 3 seconds."
- **Set performance goals:** Based on the application's requirements and user expectations, define what constitutes acceptable performance under normal and peak conditions.
  - **Example:** "The system must have a response time under 2 seconds for 95% of requests during peak usage."

Output: Document of performance goals and KPIs

- **Scenario:** "System should handle 1,000 concurrent users without response time exceeding 3 seconds."
- **Performance goal:** "95% of requests should be processed in under 2 seconds."

Step 2: Determine performance testing criteria

- **Identify performance testing types:** Decide on the types of performance tests that will be needed (e.g., load testing, stress testing, scalability testing).
  - **Example:** "We will conduct load testing to simulate up to 1,000 users and stress testing to understand how the system performs under extreme conditions."
- **Define the system behavior under various load conditions:** Plan for how much load (e.g., concurrent users) will be applied and for how long.
  - **Example:** "Start with 100 users and gradually increase to 1,000 users over 1 hour."

Output: Testing plan

- **Test type:** Load testing and stress testing.
- **Load conditions:** "Start with 100 users and increase to 1,000 users over 1 hour."

Step 3: Prepare test environment

- **Set up test environment:** Ensure that the test environment closely mirrors the production environment. This includes the hardware, software, and network configurations.
  - **Example:** "Set up a staging environment with the same server configurations and databases as the production environment."
- **Prepare test data:** Create realistic data that simulates user behavior and system usage, such as test user profiles, product catalogs, and transactions.
  - **Example:** "Create 1,000 test user accounts and populate the product catalog with 500 items."

Output: Test environment setup

- **Test environment:** Staging environment mirroring production.
- **Test data:** 1,000 test users, 500 items in the product catalog.

Step 4: Select performance testing tools

- **Choose appropriate tools:** Select tools that can simulate the desired user load and capture performance data.
  - **Examples of tools:** Apache JMeter, LoadRunner, Gatling, or BlazeMeter.

Output: Tool selection

- **Tool:** Apache JMeter.
- **Purpose:** Simulate up to 1,000 concurrent users and measure response time.

Step 5: Design performance test scenarios

- **Develop test cases:** Create test scenarios that reflect real-world usage. These could involve tasks like browsing, logging in, making purchases, etc.

© NIMI NOT TO BE REPUBLISHED

- **Example:** "Simulate 80% of users browsing products, 10% adding products to their cart, and 10% completing a purchase."

- **Define load distribution:** Specify the number of virtual users that will perform each action in the system.
  - **Example:** "For 1,000 users, simulate 800 browsing, 100 adding to the cart, and 100 purchasing."

Output: Performance test cases

- **Scenario 1:** 800 users browsing products.
- **Scenario 2:** 100 users adding products to the cart.
- **Scenario 3:** 100 users completing a purchase.

Step 6: Execute the performance test

- **Run tests:** Start the test by gradually increasing the load according to the defined load distribution. For load testing, start small and incrementally increase the number of users.
  - **Example**: "Simulate 100 users at first, then increase by 100 users every 10 minutes until reaching 1,000 users."
- **Monitor system performance:** While the test is running, monitor the system's key performance metrics such as
  - Response times.
  - CPU and memory usage.
  - Error rates.
  - Throughput (number of requests processed per unit of time).
  - **Example:** "At each load level, monitor response times, CPU, and memory utilization."

Output: Performance test execution data

- **At 100 users:** Response time = 1.5 seconds, CPU = 45%, Memory = 60%, No errors.
- **At 500 users:** Response time = 2.2 seconds, CPU = 65%, Memory = 70%, No errors.
- **At 1,000 users:** Response time = 3.0 seconds, CPU = 85%, Memory = 80%, 1% error rate (timeouts).

Step 7: Analyze the performance test results

- **Evaluate against goals:** Compare the actual system performance against the defined benchmarks and KPIs.
  - **Example:** "At 1,000 users, response time exceeded the goal of 2 seconds (actual was 3.0 seconds). CPU usage is high at 85%, which could indicate a bottleneck."
- **Identify performance bottlenecks:** Review system logs, error rates, and resource utilization to pinpoint any weaknesses in the system.
  - **Example:** "CPU usage spikes as the number of users increases, suggesting a potential bottleneck in server capacity."

Output: Performance test analysis

- **Response time at 1,000 users:** 3.0 seconds (exceeds benchmark).
- **CPU usage at 1,000 users:** 85% (suggests potential server overload).
- **Memory usage at 1,000 users:** 80% (acceptable).

Step 8: Identify bottlenecks and issues

- **Investigate issues:** Based on the test results, determine the root causes of performance degradation, such as database queries, server capacity, or inefficient code.
  - **Example:** "The system is experiencing high CPU usage at peak load, which may be due to inefficient database queries or insufficient server resources."
- **Profile the system:** Use profiling tools to dig deeper into system resource usage.
  - **Example:** "Use database profiling to identify slow queries that may be contributing to performance issues."

Output: Bottleneck report

- **Issue 1:** High CPU usage at peak load (85%).
- **Issue 2:** Slow database queries affecting response times.
- **Issue 3:** Network latency observed in certain transactions.

Step 9: Optimize the system and re-test

- **Make improvements:** Work with the development and operations teams to optimize the system, such as optimizing database queries, adding more servers, or improving code efficiency.
  - **Example:** "Optimize slow database queries and add more server resources to handle the load."
- **Retest after optimizations:** Once the issues are fixed, rerun the performance tests to confirm that the optimizations have improved the system's performance.
  - **Example:** "Re-run the test with 1,000 users after optimizing the database queries and adding more servers."

Output: Retest results after optimization

- **After optimization:** Response time = 2.5 seconds at 1,000 users, CPU = 70%, no errors.

Step 10: Generate final performance testing report

- Create a comprehensive report: Document all the test scenarios, execution results, issues, root causes, optimizations made, and final performance metrics.

© NIMI NOT TO BE REPUBLISHED

- **Example:** "The system successfully supports 1,000 concurrent users after optimizations. Response time improved from 3.0 to 2.5 seconds."

- **Share the report with stakeholders:** Share the results with stakeholders, such as project managers, developers, and clients, for review and final approval.

Output: Final performance testing report

- **Objective:** Test system performance with 1,000 concurrent users.
- **Test results:** Response time of 3.0 seconds at 1,000 users, CPU usage of 85%, 1% errors.
- **Optimizations:** Improved database queries, added server resources.
- **Final results:** Response time of 2.5 seconds at 1,000 users, CPU usage reduced to 70%, no errors.

Example of output scenario

1 Performance test execution

- **At 100 users:** Response time = 1.5 seconds, CPU usage = 45%.
- **At 500 users:** Response time = 2.2 seconds, CPU usage = 65%.
- **At 1,000 users:** Response time = 3.0 seconds, CPU usage = 85%, 1% errors.

2 Issues identified

- **Issue 1:** Response time exceeded the goal at 1,000 users (3.0 seconds).
- **Issue 2:** High CPU usage at peak load (85%).

3 Optimizations made

- Database query optimization to speed up data retrieval.
- Increased server resources to distribute load.

4 Retest results

- **After optimization:** Response time = 2.5 seconds at 1,000 users, CPU usage = 70%, no errors.

© NIMI
NOT TO BE REPUBLISHED

Diff. b/w above 3

Objective: At the end of this exercise you shall be able to
- **stress tests extreme conditions, load tests normal, performance measures responsiveness.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with test management tools	- as reqd.

PROCEDURE

Performance Testing, Load Testing, and Stress Testing are all subsets of Performance Testing, but each focuses on different aspects of a system's behavior. Below is a step-by-step breakdown of the differences, along with outputs that might arise in each testing type.

1 Performance testing

- **Goal:** The goal of Performance Testing is to determine how well the system performs under varying conditions, focusing on response times, throughput, and resource utilization (like CPU, memory, etc.). It encompasses various types of testing (e.g., load, stress, and scalability tests) to assess the system's performance.

Key characteristics

- Measures system performance according to set criteria.
- Involves multiple types of tests, including Load Testing and Stress Testing.
- Results are compared against defined benchmarks and goals.

Example of performance testing steps

i **Objective:** Test how the system behaves under normal load (e.g., 1,000 concurrent users).

ii **Define metrics:** Response time ≤ 2 seconds, CPU usage ≤ 80%.

iii **Execute test:** Run the system with 1,000 concurrent users.

iv **Output:** Response time = 2.1 seconds, CPU usage = 75%, no errors.

Output (Performance testing)

- **Result:** "The system meets performance benchmarks with no significant issues."
- **Analysis:** "CPU usage is within acceptable limits. Response times are slightly over the 2-second target but still within a tolerable range."

2 Load testing

- **Goal:** Load Testing focuses on determining how the system handles expected user loads. It measures the system's response to the anticipated amount of traffic (e.g., 1,000 users) and ensures that performance meets predefined expectations under normal conditions.

Key characteristics

- Aims to simulate the expected load or typical user traffic.
- Measures performance against defined goals (e.g., response time, throughput).
- Test load gradually increases until the expected load is reached.

Example of load testing steps

i **Objective:** Test the system's ability to handle 1,000 concurrent users.

ii **Define metrics:** Response time should be ≤ 2 seconds for 95% of requests.

iii **Execute test:** Gradually increase load from 100 to 1,000 users over 1 hour.

iv **Output**
- **At 100 users:** Response time = 1.5 seconds, CPU usage = 60%, no errors.
- **At 500 users:** Response time = 2 seconds, CPU usage = 70%, no errors.
- **At 1,000 users:** Response time = 2.5 seconds, CPU usage = 80%, no errors.

Output (Load testing)

- **Result:** "The system handles 1,000 concurrent users with a response time under 3 seconds."
- **Analysis:** "The response time exceeds the target slightly at 1,000 users. CPU usage remains within an acceptable range."

3 Stress testing

- **Goal:** Stress Testing evaluates the system's behavior under extreme or unpredictable loads, pushing the system beyond its capacity to see how it breaks, recovers, or fails. It is often used to identify the system's breaking point.

Key characteristics

- Tests the system under stressful conditions or extreme scenarios (e.g., 10,000 concurrent users).
- Focuses on identifying the system's limits (e.g., maximum number of concurrent users it can handle before failure).
- Measures system recovery post-failure.

© NIMI NOT TO BE REPUBLISHED

Example of stress testing steps

i **Objective:** Test how the system behaves with 10,000 concurrent users.

ii **Define metrics:** Response time should be < 5 seconds up to the breaking point.

iii **Execute test:** Gradually increase users until system failure or breakdown occurs.

iv **Output**

- **At 1,000 users:** Response time = 2.5 seconds, CPU usage = 80%, no errors.
- **At 5,000 users:** Response time = 5 seconds, CPU usage = 95%, minor errors.
- **At 10,000 users:** Response time = 10 seconds, CPU usage = 100%, system crashes.

Output (Stress testing)

- **Result:** "The system fails to handle 10,000 users, with a crash occurring at this load."
- **Analysis:** "The system's breaking point is around 7,000 concurrent users. Recovery time from failure is slow."

Summary of differences

Aspect	Performance Testing	Load Testing	Stress Testing
Objective	General testing of system performance.	Test system's ability to handle expected load.	Test system's behavior under extreme load conditions.
Scope	Broad, includes multiple tests like load, stress.	Simulate expected user load and behavior.	Identify breaking point and system failure behavior.
Testing Condition	Varies (could be normal or stress conditions).	Normal load conditions (e.g., 1,000 users).	Extreme load conditions (e.g., 10,000 users or more).
Focus	Meets performance benchmarks under various conditions.	Performance under expected load.	Performance under extreme stress; system recovery.
Expected Outcome	System meets defined KPIs (response time, throughput).	System meets performance expectations under normal load.	System's behavior under overload and recovery.
Example of Output	Response time ≤ 2 seconds, CPU ≤ 80% at 1,000 users.	Response time ≤ 2.5 seconds, CPU ≤ 85% at 1,000 users.	System crashes at 10,000 users, recovery is slow.

Step-by-step comparison with output

Performance testing

- **Objective:** Test if the system meets performance criteria (response time, throughput).
- **Test conditions:** Varies between load, stress, and scalability.
- **Example test results**
  - **At 100 users:** Response time = 1.5 seconds, CPU = 60%.
  - **At 500 users:** Response time = 2.0 seconds, CPU = 70%.
  - **At 1,000 users:** Response time = 2.5 seconds, CPU = 80%.
- **Conclusion:** "The system performs within acceptable limits but needs optimization for larger loads."

Load testing

- **Objective:** Test how the system handles expected traffic (e.g., 1,000 users).
- **Test conditions:** Gradual increase of users until expected load is reached.
- **Example test results**
  - **At 100 users:** Response time = 1.5 seconds.
  - **At 500 users:** Response time = 2.0 seconds.
  - **At 1,000 users:** Response time = 2.5 seconds, CPU = 80%.
- **Conclusion:** "The system can handle 1,000 concurrent users, but some optimizations are needed for better performance."

Stress testing

- **Objective:** Test the system's breaking point and recovery under extreme load.
- **Test conditions:** Gradual increase of users beyond expected load until system failure.
- **Example test results**
  - **At 1,000 users:** Response time = 2.5 seconds, CPU = 80%.
  - **At 5,000 users:** Response time = 5.0 seconds, CPU = 95%.
  - **At 10,000 users:** System crashes, response time = 10 seconds, CPU = 100%.
- **Conclusion:** "The system crashes at 10,000 users and needs significant improvements to handle high loads."

© NIMI NOT TO BE REPUBLISHED

The benefits of automated testing

Objective: At the end of this exercise you shall be able to
- **automated testing improves efficiency, accuracy, and consistency.**

Requirements

Tools/Equipment/Machines

- A working PC with Faster Test Execution - as reqd.

PROCEDURE

Automated testing offers significant benefits compared to manual testing, particularly in terms of speed, accuracy, and coverage. Below, we will explore the benefits of automated testing step by step, along with the expected outcomes at each stage.

Step 1: Faster Test Execution

Benefit: Automated tests are faster to execute than manual tests, as they can run continuously without human intervention.

How it Works

- Automated tests can run much faster, especially when dealing with large test suites. For instance, the same tests that might take several days manually can be completed in hours using automation.
- Automated tests can be executed repeatedly in parallel (on multiple machines, environments, or browsers) without manual intervention.

Example

- **Manual Testing:** A set of 100 tests may take 3 hours for a tester to complete manually.
- **Automated Testing:** The same set of 100 tests can be executed in 15 minutes through automation.

Output

- **Manual Testing Time:** 3 hours.
- **Automated Testing Time:** 15 minutes.

Step 2: Increased Test Coverage

Benefit: Automated testing allows for broader and deeper test coverage that would be difficult or time-consuming to achieve manually.

How it Works

- You can execute hundreds or even thousands of tests across different browsers, devices, and environments in a fraction of the time it would take to do manually.
- Test cases, including edge cases, that might otherwise be skipped due to time constraints can now be executed.

Example

- **Manual Testing:** A tester might only focus on the most critical paths (e.g., login, basic navigation), leaving many parts of the system untested.
- **Automated Testing:** An automated test suite can run through all critical paths and additional tests like boundary conditions, database tests, and cross-platform tests.

Output

- **Manual Testing Coverage:** 10 out of 20 test scenarios.
- **Automated Testing Coverage:** 20 out of 20 test scenarios, including edge cases and cross-platform tests.

Step 3: Improved Accuracy and Consistency

Benefit: Automated testing eliminates the possibility of human errors, ensuring that tests are executed consistently every time.

How it Works

- Once automated tests are written, they are executed with the same steps and parameters, leading to consistent results.
- In manual testing, testers might make mistakes, miss steps, or fail to perform tests the same way across different sessions.

Example

- **Manual Testing:** A tester might forget to check a button's functionality or click it in the wrong sequence.
- **Automated Testing:** The test script will always perform the same steps, ensuring accuracy in execution.

© NIMI NOT TO BE REPUBLISHED

Output

- **Manual Testing Results:** Error in test execution due to inconsistent manual steps.
- **Automated Testing Results:** Consistent and repeatable results, with no missed steps.

Step 4: Cost-Effective in the Long Run

Benefit: Although the initial investment in automated testing tools and scripts can be high, the long-term cost savings can be significant, especially for regression testing.

How it Works

- Once test scripts are written and automated, they can be reused in future sprints or releases, reducing the need for repetitive manual work.
- Automated tests can run on different environments simultaneously, saving time and effort in cross-browser testing, system integration testing, and regression testing.

Example

- **Manual Testing:** If each test cycle costs $5,000 due to manual labor, over 10 iterations, the total cost would be $50,000.
- **Automated Testing:** Initial setup for automated testing costs $15,000. Afterward, running tests for each cycle costs only $1,000 in maintenance, leading to significant cost savings.

Output

- **Manual Testing Total Cost:** $50,000.
- **Automated Testing Total Cost:** $15,000 (initial) + $10,000 (10 cycles of maintenance) = $25,000.

Step 5: Early Detection of Defects

Benefit: Automated tests can be run continuously during development (e.g., during a Continuous Integration pipeline), which leads to early detection of defects.

How it Works

- Automated tests can be integrated into the CI/CD pipeline and run every time new code is committed. This allows the team to identify defects immediately and fix them before they progress too far.
- In manual testing, the test cycle may occur later in the development process, meaning issues are found later when they are harder to address.

Example

- **Manual Testing:** Defects may be detected only during the testing phase after development is complete.
- **Automated Testing:** Tests run after every code change, quickly catching defects as soon as they occur.

Output

- **Manual Testing:** Issues detected 2 weeks after development.
- **Automated Testing:** Issues detected within hours of code commit.

Step 6: Enhanced Reusability of Test Scripts

Benefit: Once written, automated test scripts can be reused across multiple versions, releases, and environments.

How it Works

- Automated tests can be reused with minimal changes. Test scripts can be executed across different platforms or configurations, and they can be adapted for new versions of the application with minimal changes.
- Test scripts are also useful for performing regression testing each time a new feature is added or the system is updated.

Example

- **Manual Testing:** The test suite needs to be recreated for each new version, taking significant time.
- **Automated Testing:** The same test scripts can be reused in future versions, and only minor changes might be needed to accommodate new functionality.

Output

- **Manual Testing:** 50 test cases need to be recreated for each new version.
- **Automated Testing:** 50 test cases are reused with minor adjustments for the new version.

Step 7: Testing in Parallel Across Multiple Environments

Benefit: Automated testing allows testing on multiple platforms, browsers, or devices at the same time, ensuring the application works across different configurations.

How it Works

- Automated tests can run in parallel across multiple environments (e.g., different browsers, OS versions, devices) without requiring manual intervention. This is particularly useful for cross-browser or cross-platform testing.
- Manual testing would require testing each configuration separately, which takes much longer.

Example

- **Manual Testing:** Testing the same application on multiple browsers (Chrome, Firefox, Safari) requires separate tests for each.
- **Automated Testing:** Run the same tests across all browsers simultaneously.

© NIMI NOT TO BE REPUBLISHED

Output

- **Manual Testing:** 1 test for each browser (3 total), each taking time sequentially.
- **Automated Testing:** 1 test running in parallel across Chrome, Firefox, and Safari, all finishing simultaneously.

Step 8: Easy Integration with Continuous Integration (CI) and Continuous Deployment (CD) Pipelines

Benefit: Automated tests can be integrated into the CI/CD pipeline, ensuring that tests are executed every time code changes are committed, leading to faster deployment cycles.

How it Works

- Automated tests can be triggered automatically as part of the build process in the CI/CD pipeline. This integration helps to ensure that code quality is maintained as new features are added and code is deployed to production.
- In manual testing, tests are often delayed and may be skipped, slowing down the release cycle.

Example

- **Manual Testing:** Tests are executed after code is completed, delaying deployment.
- **Automated Testing:** Tests are triggered every time code changes are committed, allowing for faster releases.

Output

- **Manual Testing:** Deployment cycles delayed by 3–5 days for manual testing.
- **Automated Testing:** Deployment cycles reduced to 1–2 days, with testing integrated into CI/CD.

Step 9: Better Reporting and Traceability

Benefit: Automated tests generate logs and reports that help developers track test results and identify failures quickly.

How it Works

- Automated tests provide detailed reports with logs of each test executed. This includes data on pass/fail results, error messages, and the stack trace for failures.
- Manual testing relies on human-recorded logs and may miss important details, reducing the traceability of defects.

Example

- **Manual Testing:** A tester manually records the results, which might miss critical details.
- **Automated Testing:** Detailed logs and reports are generated, including execution time, pass/fail results, and failure stack traces.

Output

- **Manual Testing:** Limited report with potential errors due to missed steps.
- **Automated Testing:** Detailed report with logs, error messages, and failure reasons.

© NIMI NOT TO BE REPUBLISHED

Understanding the testing process

Objective: At the end of this exercise you shall be able to
- **winrunner automates test execution, enhancing testing efficiency.**

Requirements

Tools/Equipment/Machines

- A working PC with Quality Center (HP ALM) - as reqd.

PROCEDURE

Understanding the testing process is key to ensuring high-quality software delivery. Testing involves systematically checking a system or application for defects, inconsistencies, and potential issues. Below is a **step-by-step guide to the software testing process,** with expected outcomes at each stage.

Step 1: Requirements Analysis

Goal: Understand the application's requirements and specifications to define what needs to be tested.

How it Works

- Review the project documentation, user stories, business requirements, and functional specifications to gather information on the expected behavior of the software.
- Identify the testable requirements, expected results, and constraints of the system.

Example

- **Input:** Business requirements document, user stories.
- **Action:** Break down the requirements to understand features like login functionality, user permissions, or payment processing.

Output

- **Testable Requirements**
  - User should be able to log in using email and password.
  - Payment should be processed securely.

Conclusion: The testing team has a clear understanding of what features need to be tested.

Step 2: Test Planning

Goal: Define the scope, strategy, and resources needed to perform testing.

How it Works

- Develop a test plan that outlines the objectives, testing scope, timelines, testing approach (manual vs. automated), team roles, and tools used for testing.
- Define the entry and exit criteria, testing types (e.g., functional, non-functional, regression), and risks associated with testing.

Example

- **Input:** Requirements document, team availability, test tools.
- **Action:** Define test scope (e.g., testing login functionality, payment gateway) and strategy (e.g., manual testing, automated tests for regression).

Output

- **Test Plan Document**
  - **Scope:** Functional and regression testing.
  - **Tools:** Selenium for automated tests, JIRA for bug tracking.
  - **Timeline:** 2 weeks for initial testing, 1 week for re-testing.

Conclusion: The test team is ready with a clear plan to execute testing.

Step 3: Test Case Design

Goal: Design detailed test cases and scenarios based on the requirements.

How it Works

- Based on the test plan, the team creates specific test cases and scenarios that describe how the functionality should be validated.
- Each test case defines the test input, expected output, and steps to follow. The design may involve positive and negative tests, boundary conditions, and edge cases.

Example

- **Input:** Test cases for login functionality.
- **Action:** Define test cases such as:

© NIMI NOT TO BE REPUBLISHED

- **Test Case 1:** Valid login with correct email and password.
- **Test Case 2:** Invalid login with incorrect password.
- **Test Case 3:** Login attempt with empty email field.
- **Test Case 4:** Login attempt with empty password field.
- **Test Case 5:** Login attempt with an unregistered email address.

Output

- **Test Case Document**
  - **Test Case 1**
    - i **Steps:** Enter valid email, enter correct password, click login.
    - ii **Expected Result:** User successfully logs in.
  - **Test Case 2**
    - i **Steps:** Enter valid email, enter incorrect password, click login.
    - ii **Expected Result:** Error message displayed (Invalid password).

Conclusion: The team has well-defined test cases to run during testing.

Step 4: Test Environment Setup

Goal: Prepare the testing environment to execute the tests.

How it Works

- Set up the required hardware, software, network configurations, databases, and tools needed for testing.
- This environment can either mirror the production environment or be a separate dedicated testing setup, depending on the project's needs.

Example

- **Input:** Required hardware/software specifications.
- **Action:** Install the application to be tested, configure the test environment (e.g., set up test servers, configure test databases, and deploy test tools).

Output

- **Test Environment Ready**
  - Test environment set up with correct OS version, browser configurations, and database setup.
  - Tools like Selenium, JIRA, and test data generators are ready.

Conclusion: The testing environment is prepared and all necessary configurations are complete.

Step 5: Test Execution

Goal: Execute the test cases and record the results.

How it Works

- Test cases are executed manually or using automation tools, depending on the plan. During execution, results are recorded for later analysis.
- Defects are identified during this phase, and the testing team tracks any failures.

Example

- **Input:** Test cases and testing environment.
- **Action:** Run the login test cases using the defined steps, using either a test execution tool or manually interacting with the system.

Output

- **Test Execution Results**
  - **Test Case 1 (Valid login):** Passed.
  - **Test Case 2 (Invalid login):** Failed (Error message not displayed).
  - **Test Case 3 (Empty email):** Passed.
  - **Test Case 4 (Empty password):** Failed (Error message not displayed).

Conclusion: The testing team identifies that there are issues with error messages for invalid login attempts.

Step 6: Defect Reporting

Goal: Log and track defects found during test execution.

How it Works

- When a test case fails, it is recorded as a defect. A detailed bug report is created with steps to reproduce the defect, the environment in which it was found, and any other relevant details.
- Defects are tracked using tools like JIRA, Bugzilla, or other defect-tracking systems.

Example

- **Input:** Test case failure (failed error message display).
- **Action:** Log the defect in a defect-tracking tool with detailed information (e.g., "Login page does not display error for incorrect password").
- **Details**
  - **Defect ID:** 12345
  - **Description:** Error message for invalid password not displayed.
  - **Severity:** Critical.
  - **Steps to reproduce:** Enter a valid email, enter incorrect password, click login.

Output

- **Defect Report**
  - **Defect ID:** 12345
  - **Status:** Open.

© NIMI NOT TO BE REPUBLISHED

- **Severity:** Critical.
- **Assigned to:** Developer for fixing.

Conclusion: The defects have been documented, and the development team will start working on fixing them.

Step 7: Defect Fixing

Goal: Developers fix the defects identified during testing.

How it Works

- Developers analyze the defect reports, identify the root cause of issues, and implement fixes to resolve the defects.
- Once fixes are made, the software is sent back to the testing team for verification.

Example

- **Input:** Defect report for error message failure.
- **Action:** The developer investigates the issue and modifies the error-handling logic in the login module.

Output

- **Defect Fixed:** The error message issue is resolved, and the system now displays the correct error message for invalid login attempts.

Conclusion: The defects are fixed, and the software is ready for re-testing.

Step 8: Re-Testing and Regression Testing

Goal: Ensure that the defect fix did not introduce new issues and that the software still works as expected.

How it Works

- Re-test the failed test cases to verify that the issues have been fixed.
- Perform regression testing to ensure that the recent changes did not affect existing functionality.
- Automated testing tools are particularly useful here to quickly run a suite of tests across various scenarios.

Example

- **Input:** Fixed software.
- **Action:** Re-run the login tests to confirm the error message is now displayed correctly. Run regression tests to confirm other features are working.

Output

- **Re-Test Results**
  - **Test Case 2 (Invalid login):** Passed (Error message now displayed).
  - **Regression Test Results:** All previous tests pass.

Conclusion: The defect has been successfully fixed, and no new issues were introduced.

Step 9: Test Closure

Goal: Close the testing phase and deliver results to stakeholders.

How it Works

- Once all test cases are executed and defects have been addressed, the testing team prepares a test summary report.
- The report includes test execution results, defect status, and an assessment of the software's quality.
- The team then communicates the findings to stakeholders for decision-making (e.g., release or delay).

Example

- **Input:** Test execution results, defect status.
- **Action:** Prepare a final test report summarizing the testing activities, outcomes, and any unresolved issues.

Output

- **Test Closure Report**
  - **Test Cases Executed:** 50.
  - **Passed:** 45.
  - **Failed:** 5 (fixed and re-tested).
  - **Critical Issues:** 0.

Conclusion: The testing phase is closed, and the software is ready for release.

© NIMI NOT TO BE REPUBLISHED

Exploring the WinRunner Window

Objective: At the end of this exercise you shall be able to
- **explore WinRunner window for test creation and execution.**

Requirements

Tool/Instruments

- A working PC with User Interface (UI) Tools - as reqd.

PROCEDURE

WinRunner is an automated functional testing tool that allows users to record and run tests for software applications, mainly in GUI-based systems. It's often used for regression testing, performance testing, and other functional testing scenarios.

Below is a step-by-step guide to **exploring the WinRunner window**, explaining the different components, functionalities, and expected outputs at each stage.

Step 1: Launching WinRunner

Goal: Open the WinRunner application and get familiar with its interface.

How it Works

1 **Launch WinRunner** by clicking on the icon from the desktop or the start menu.

2 Upon launching, the **WinRunner window** will open with multiple tabs and panels.

Expected Output

- **WinRunner Main Window** with several sections:
  - Editor Window (for script creation)
  - Start/Stop Recording button (to begin recording user actions)
  - Test Script and Results tab (for viewing and running tests)
  - Command Button Area (to run and manage tests)

Step 2: Exploring the Main Interface

Goal: Familiarize yourself with the key sections of the WinRunner interface.

How it Works

1 **Menu Bar:** Located at the top of the WinRunner window, it contains various options such as File, Edit, View, and Tools.

2 **Toolbars:** The toolbars are visible beneath the menu bar and provide quick access to key features, such as starting a test, inserting checkpoints, or stopping the script.

3 **Editor Window:** The central area where scripts are created and modified. It displays the code for automated tests.

4 **Test Script & Results Tab:** Shows the status of the test and results after execution.

5 **Playback Controls:** Located at the top, used for controlling the test execution (Play, Stop, Pause).

Expected Output

- **WinRunner UI Layout**
  - Menu Bar: **File | Edit | Tools | Options.**
  - Toolbars with buttons for **Record, Play, Stop.**
  - **Editor Window:** Blank for script creation.
  - **Test Script Tab** and **Results Tab** are visible.

Step 3: Recording a Test Script

Goal: Record a user interaction (such as a login) to create a test script.

How it Works

1 Click the **Start Recording** button (typically found in the toolbar).

2 Perform the actions in the application you want to test (e.g., login to an app).

3 Once the actions are recorded, click **Stop Recording**.

Expected Output

- WinRunner automatically generates a test script based on the actions performed.
  - **Example Script (Fig 1)**

Fig 1

```
window_activate("LoginWindow");
edit_set("username_field", "testuser");
edit_set("password_field", "password123");
button_press("login_button");
```

© NIMI NOT TO BE REPUBLISHED

- **Script is displayed in the Editor Window** for review and modification.

Step 4: Inserting Checkpoints

Goal: Insert checkpoints to verify that specific conditions are met during the test execution.

How it Works

1 Position the cursor at the point in the script where you want to insert the checkpoint.

2 Click on the Insert Checkpoint button in the toolbar.

3 Choose the appropriate checkpoint type (e.g., GUI Checkpoint, Bitmap Checkpoint, Text Checkpoint).

4 Select the object or condition you want to verify (e.g., verify that a label displays the correct text).

Expected Output

- **Checkpoints Inserted**
  - **GUI Checkpoint Example (Fig 2)**

Fig 2

```
gui_check("window", "WindowTitle", 0, "ok_button");
```

  - This verifies that the window with the specified title exists and the "OK" button is displayed.
- **Test Script Updated** with the checkpoint.

Step 5: Running the Test Script

Goal: Execute the recorded script to verify its behavior.

How it Works

1 Click on the Play button in the toolbar to run the test.

2 WinRunner will execute the script and simulate the user actions (e.g., login, button clicks).

3 During the execution, the Test Script & Results Tab will show the test status (running, passed, or failed).

Expected Output

- **Execution Status** in the Results Tab:
  - **Pass:** If the script runs without issues and the checkpoint conditions are met.
  - **Fail:** If there are any issues (e.g., incorrect password, missing button).
- **Log Output**
  - **Example Log (Fig 3)**

Fig 3

```
Executing Script...
Login window opened.
Entering username...
Entering password...
Login successful.
Test passed.
```

Step 6: Debugging Test Scripts

Goal: Troubleshoot and correct any issues within the test script.

How it Works

1 If the test fails, WinRunner will show the error details in the Test Script & Results Tab.

2 Click on the Debug button in the toolbar or use the Debugger option in the menu to step through the script and identify issues.

3 Modify the script to fix errors (e.g., incorrect object names, missing steps).

Expected Output

- **Error Identification** in the **Results Tab**
  - **Example Error:** "Button not found" or "Incorrect value entered."
- **Modified Test Script**
  - After identifying the problem, you might need to fix object names or test actions. (Fig 4)

Fig 4

```
button_press("login_button");  // Fixed button object name
```

Step 7: Saving the Test Script

Goal: Save the test script for future executions and analysis.

How it Works

1 Click File from the menu bar and choose Save or Save As.

2 Name the script and specify the location where you want to store the test file.

Expected Output

- **Test Script Saved** in the specified location (e.g., login_test.tcl).
- The script can now be executed anytime or modified as needed.

Step 8: Analyzing Test Results

Goal: Review the results of the test execution to verify the software's functionality.

How it Works

1 After running the test, check the **Results Tab** for a summary of the test run.

2 WinRunner will show
  - **Pass/Fail Results** for each step in the script.
  - **Logs** with detailed information about the execution.

Expected Output

- **Test Results Summary**
  - **Passed:** The login test completed successfully.
  - **Failed:** Error in the text box verification.

© NIMI NOT TO BE REPUBLISHED

- **Detailed Log (Fig 5)**

Fig 5

```
Test Results:
---------------------
Step 1: Pass - Login window appeared.
Step 2: Fail - Text box did not display expected text.
```

Step 9: Generating Reports

Goal: Generate and export test execution reports for analysis and sharing with stakeholders.

How it Works

1 After test execution, click on Report in the toolbar or go to the Tools menu.

2 Choose the type of report you want to generate (e.g., Summary Report, Detailed Report).

3 Save or export the report in your desired format (e.g., HTML, PDF).

Expected Output

- **Generated Report** with test execution results, error logs, and other relevant information.
  - **Example Report Output (Fig 6)**

Fig 6

```
Test Name: Login Functionality
Test Status: Passed
Duration: 00:05:30
Issues: None
```

Step 10: Closing WinRunner

Goal: Close the WinRunner application and ensure all work is saved.

How it Works

1 Close the Test Script window.

2 Choose Exit from the File menu.

3 If there are unsaved changes, WinRunner will prompt you to save the test script.

Expected Output

- **WinRunner Closed** without losing any progress.
- Test scripts are saved and reports are generated for future reference.

© NIMI
NOT TO BE REPUBLISHED

How does WinRunner identify GUI objects

Objective: At the end of this exercise you shall be able to
- **identifies GUI objects through unique properties.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

WinRunner uses a process called **GUI (Graphical User Interface) Object Recognition** to identify and interact with elements on the screen. This allows WinRunner to simulate user interactions by recognizing objects such as buttons, text fields, menus, and checkboxes.

Here is a **step-by-step guide** on how **WinRunner identifies GUI objects**, with expected outputs at each stage:

Step 1: Launch WinRunner and Load the Application

Goal: Start WinRunner and load the application under test.

How it Works

1 Open WinRunner and start recording by clicking the Record button.

2 Open the application you want to test (e.g., a login page, a form, or a button).

Expected Output

- **WinRunner UI** opens with the **Test Script** tab.
- **Application** is launched and is ready for user interactions.

Step 2: Object Identification Using WinRunner's GUI Map

Goal: Identify the objects (buttons, text fields, etc.) within the application using GUI Map.

How it Works

1 **GUI Map:** WinRunner uses a GUI Map to store and identify objects within the application.

2 **Object Properties:** Objects are identified based on their unique properties (like the name, label, or type) within the application.

Example

- **Object Properties** can be:
  - **Name:** A unique identifier for the object.
  - **Type:** Button, Text Field, Checkbox, etc.
  - **Window:** Parent window where the object resides.
  - **Value:** Any text or value associated with the object.

Expected Output

- **GUI Map** is populated with the objects of the application, such as:
  - **Object:** "Login Button"
  - **Type:** Button
  - **Properties:** Name = "login_button", Type = "PushButton", Window = "LoginWindow"

Step 3: Object Recognition During Recording

Goal: WinRunner recognizes and records objects as the user interacts with them.

How it Works

1 When you click on a button or enter text in a field, WinRunner records the actions and identifies the object under the cursor.

2 WinRunner looks for unique identifiers like the object name, type, or value that makes it distinct.

3 WinRunner records the action as a command in the test script.

Example

- When you click the "Login" button, WinRunner detects it as an object and records the interaction as:

Expected Output (Fig 1)

Fig 1

```
button_press("login_button");
```

- **Recorded Script (Fig 2)**

Fig 2

```
window_activate("LoginWindow");
edit_set("username_field", "testuser");
edit_set("password_field", "password123");
button_press("login_button");
```

- The object "login_button" is recognized based on its properties like the **name** and **type.**

© NIMI NOT TO BE REPUBLISHED

Step 4: Using Object Spy for Manual Object Identification

Goal: Manually identify objects if they are not recorded automatically.

How it Works

1 If WinRunner doesn't automatically detect an object, you can use the Object Spy tool.

2 Object Spy helps you to point to an element on the screen and collect its properties.

3 The tool allows you to view and capture the object properties of any visible element within the application.

Steps

1 Click on the Object Spy button (or choose it from the toolbar).

2 Point and click on the GUI object within the application (like a button or text field).

3 WinRunner will display the properties of the selected object.

Expected Output

- **Object Properties** Displayed
  - **Object Name:** "username_field"
  - **Object Type:** "Edit"
  - **Parent Window:** "LoginWindow"
  - **Additional Properties:** Class = "TextBox", Text = "Username", etc.

Step 5: Storing Objects in the GUI Map

Goal: Add recognized objects to the GUI Map for later reference.

How it Works

1 After using the Object Spy tool to capture an object's properties, WinRunner asks if you want to add it to the GUI Map.

2 WinRunner saves the object's properties (name, type, parent window, etc.) into the GUI Map for later use in test scripts.

Example

- When you use Object Spy on the "username_field" text box, WinRunner prompts to add the object to the GUI Map.
- You click Yes to add it to the map.

Expected Output

- **GUI Map Entry** for "username_field":
  - **Object Name:** "username_field"
  - **Object Type:** "Edit"
  - **Properties:** Class = "TextBox", Window = "LoginWindow", etc.

GUI Map File (Fig 3)

Fig 3

```
Object: username_field
Type: Edit
Class: TextBox
Window: LoginWindow
```

Step 6: Running the Test

Goal: Execute the recorded script to verify that WinRunner can correctly interact with the identified objects.

How it Works

1 When the test script is played, WinRunner uses the GUI Map to identify and interact with the objects defined in the script.

2 WinRunner checks for the object properties (name, type, class) to ensure the correct object is being interacted with.

Example

- When executing the script, WinRunner checks the **GUI Map** for an object named **"username_field"** of type **Edit** and class **TextBox** within the **LoginWindow**.
- It then enters the username value as specified in the test script.

Expected Output

- **Execution Log (Fig 4)**

Fig 4

```
Activating Login Window...
Entering username into username_field...
Entering password into password_field...
Pressing the login_button...
Test Passed.
```

Step 7: Handling Dynamic Objects

Goal: Address situations where GUI objects are dynamic (e.g., object names change during runtime).

How it Works

1 If object properties change during runtime (e.g., the object name changes each time), WinRunner uses regular expressions or descriptive programming to identify the object dynamically.

2 Regular expressions allow WinRunner to match patterns in object properties rather than fixed values.

Example (Fig 5)

Fig 5

```
button_press("login_button*");
```

- If the **"login_button"** name changes dynamically to **"login_button_1"**, you can use:

© NIMI NOT TO BE REPUBLISHED

This uses a wildcard to match any object with a name that starts with "login_button".

Expected Output

- **Dynamic Object Recognition** using regular expressions. (Fig 6)

Fig 6

```
button_press("login_button*");  // Works even if the button name changes dynamically.
```

Step 8: Verifying Object Properties with Checkpoints

Goal: Ensure that the identified objects have the expected properties during test execution.

How it Works

1 Checkpoints are used to verify that an object's properties match the expected values during test execution.

2 WinRunner compares the actual properties of the object with the expected ones, such as the object's label, value, or state.

Example

- Insert a **GUI Checkpoint** to verify that the **login_button** displays the correct label. (Fig 7)

Fig 7

```
gui_check("login_button", "Text", "Login");
```

Expected Output

- **Checkpoint Verification**
  - If the button text matches **"Login",** the checkpoint passes.
  - If the text is incorrect, the checkpoint fails, and a report is generated.

Step 9: Handling Object Hierarchy

Goal: Work with objects that are part of a hierarchy, such as buttons within frames or windows.

How it Works

1 WinRunner can recognize objects within nested windows (e.g., buttons inside a dialog box or a frame).

2 You must specify the parent-child relationship for objects to accurately identify them.

Example

- The **OK button** is inside a dialog window called **"DialogBox"**: (Fig 8)

Fig 8

```
button_press("DialogBox.ok_button");
```

Expected Output

- **Correct Object Recognition in Hierarchy**
  - The **DialogBox.ok_button** is identified and clicked, even if it is nested within other UI elements.

© NIMI NOT TO BE REPUBLISHED

Spying on GUI map mode

Objective: At the end of this exercise you shall be able to
- **spying mode captures and identifies GUI object properties.**

Requirements

Tool/Instruments
- A working PC - as reqd.

PROCEDURE

Spying on GUI Map Mode in WinRunner: Step-by-Step with Expected Output

In **WinRunner, Spying on GUI Objects** is a critical process used to capture the properties of objects on the screen, which helps to build the **GUI Map**. This process is done using the **Object Spy** tool. It allows you to manually inspect and collect object properties like names, classes, types, and other attributes that are essential for object recognition during test execution.

Here's a detailed **step-by-step guide** on **Spying on GUI Objects in GUI Map Mode**

Step 1: Launch WinRunner

Goal: Start the WinRunner tool and prepare the environment for spying on GUI objects.

How it Works

1 Double-click the WinRunner icon to launch the tool.

2 Upon launch, the WinRunner Main Window opens with several sections (Editor, Test Script Tab, etc.).

3 Make sure you have the application you want to test open and running (the application window should be visible).

Expected Output

- **WinRunner UI** opens with the **Test Script** tab, **GUI Map**, and other controls visible.
- The application is running, and you are ready to start spying on objects within it.

Step 2: Open the Object Spy Tool

Goal: Open the Object Spy tool to begin spying on objects in the application.

How it Works

1 In the **WinRunner main window,** click on the **Tools** menu from the top menu bar.

2 From the dropdown, select **Object Spy**. Alternatively, you can find the **Object Spy** icon in the toolbar and click it directly.

3 The **Object Spy** dialog window will open, allowing you to start spying on GUI elements.

Expected Output

- **Object Spy Window** appears, which includes a Spy Cursor and other relevant controls.
  - **Spy Cursor** is the pointer you will use to click on objects in the application to collect their properties.
  - **Properties of Objects** will be shown in the Object Spy window once you start spying.

Step 3: Spying on a GUI Object

Goal: Use the Spy Cursor to identify and capture the properties of GUI objects.

How it Works

1 In the Object Spy window, you will see a Crosshair Cursor or Spy Cursor that you can move around the screen.

2 Move the Spy Cursor over the object in the application you want to spy on (for example, a button, text field, or drop-down list).

3 Click on the object. The Object Spy tool will capture its properties and display them in the Object Spy window.

Example

- **Spying on a Login Button** in the application.
- Move the Spy Cursor over the **Login Button** in the application window.
- Click on it to collect the object properties (e.g., **Name, Type, Class, Label, Window Name**).

Expected Output

- **Object Properties Displayed** in the **Object Spy** window
  - **Object Type:** Button
  - **Object Name:** login_button
  - **Class:** PushButton
  - **Window:** LoginWindow
  - **Text/Label:** Login

The tool will also display the **full path** of the object in the hierarchy (parent-child relationships), such as: (Fig 1)

Fig 1

```
Object Path: LoginWindow->login_button
```

Step 4: Add the Object to the GUI Map

Goal: After spying on the object, add its properties to the GUI Map for later use.

© NIMI NOT TO BE REPUBLISHED

How it Works

1 After spying on the object and reviewing its properties in the Object Spy window, you can add the object to the GUI Map for future reference.

2 To add the object to the GUI Map, click on the Add to GUI Map button (usually available in the Object Spy window).

3 If prompted, confirm that you want to add the object to the GUI Map.

Expected Output

- The **Object** is added to the **GUI Map** with the following information:
  - **Object Name:** login_button
  - **Type:** Button
  - **Class:** PushButton
  - **Window:** LoginWindow
  - **Additional Properties:** Text/Label = Login, etc.

The GUI Map now contains an entry for the "login_button" object with its properties.

Step 5: Verify the Object in the GUI Map

Goal: Ensure that the object has been successfully added to the GUI Map.

How it Works

1 After adding the object to the GUI Map, open the GUI Map file or go to the View menu and select GUI Map.

2 The GUI Map will list all objects that have been recognized and saved during the spying process.

Expected Output

- **GUI Map Entry** for the "login_button" will appear in the GUI Map file with the following properties. (Fig 2)

Fig 2

```
Object: login_button
Type: Button
Class: PushButton
Window: LoginWindow
Label: Login
```

- The **Object Path** would also be displayed, such as: (Fig 3)

Fig 3

```
Object Path: LoginWindow->login_button
```

Step 6: Spying on Additional Objects

Goal: Repeat the process for additional objects that need to be added to the GUI Map.

How it Works

1 Continue using the Spy Cursor to capture properties of other objects in the application.

2 For each object you spy on (e.g., text fields, checkboxes, combo boxes, etc.), follow the same steps:
  - Move the Spy Cursor over the object.
  - Click to capture the properties.
  - Add the object to the GUI Map.

Expected Output

- Each object will be added to the GUI Map with its respective properties.

For example

- **Text Field Object (Fig 4)**

Fig 4

```
Object: username_field
Type: Edit
Class: TextBox
Window: LoginWindow
Label: Username
```

- **Checkbox Object (Fig 5)**

Fig 5

```
Object: remember_me
Type: Checkbox
Class: CheckBox
Window: LoginWindow
Label: Remember Me
```

Step 7: Using the GUI Map for Test Scripting

Goal: Once objects are added to the GUI Map, they can be used in your test scripts.

How it Works

1 You can now reference the objects from the GUI Map in your test scripts.

2 In the test script, you can use the object names, types, and other properties to simulate user interactions.

Example

To simulate clicking the "Login" button, use the object name as it appears in the GUI Map: (Fig 6)

Fig 6

```
button_press("login_button");  // Refers to the object added to the GUI Map
```

Step 8: Save and Close the GUI Map

Goal: Save all the objects and close the GUI Map after completing the spying process.

How it Works

1 After adding all the objects, click on File in the GUI Map window and select Save to save the changes.

2 Close the GUI Map window when you are finished.

Expected Output

- **GUI Map Saved** with all objects and their properties for later use in test scripts.

© NIMI NOT TO BE REPUBLISHED

Choosing a GUI map mode

Objective: At the end of this exercise you shall be able to
- **choose GUI map mode for object identification and management.**

Requirements	
Tool/Instruments	
• A working PC	- as reqd.

PROCEDURE

Choosing a GUI Map Mode in WinRunner: Step-by-Step with Expected Output

In **WinRunner,** the **GUI Map Mode** determines how the tool identifies and interacts with the objects in an application. It allows you to work with **descriptive programming, stored objects in the GUI Map**, or **dynamic objects.** There are different modes for object recognition, and choosing the right one is essential for the correct identification and interaction with GUI objects.

Here's a detailed **step-by-step guide** on how to **choose a GUI Map Mode** in WinRunner, along with the expected output at each stage:

Step 1: Launch WinRunner

Goal: Start WinRunner to begin working with the GUI Map.

How it Works

1 Open WinRunner by double-clicking on its icon.

2 Once launched, the WinRunner main window will appear, featuring several sections like Test Script, GUI Map, Editor, and more.

Expected Output

- **WinRunner UI** opens and is ready for you to select the appropriate GUI Map Mode.

Step 2: Open the GUI Map

Goal: Open the **GUI Map** to view or select the objects you want to work with.

How it Works

1 From the WinRunner main window, go to the View menu at the top.

2 Select GUI Map from the drop-down menu. Alternatively, you can click the GUI Map button if available in the toolbar.

Expected Output

- The **GUI Map** window opens and displays a list of objects that have been previously identified and saved in the GUI Map.

Step 3: Choose the GUI Map Mode

Goal: Choose the GUI Map Mode to specify how WinRunner will interact with objects.

How it Works

1 Go to the Tools menu in the WinRunner main window.

2 Select GUI Map Mode from the drop-down menu. This will present several modes that affect how objects are recognized and manipulated.

Available GUI Map Modes

1 Learn Mode: This mode allows WinRunner to learn and record objects from the application. It's used for adding new objects to the GUI Map.

2 Edit Mode: This mode lets you edit properties of objects that are already recorded in the GUI Map.

3 Playback Mode: In this mode, WinRunner uses the GUI Map objects for playback during automated testing.

4 Descriptive Programming Mode: Allows you to use descriptive programming, where objects are referenced based on their properties dynamically during runtime.

Example

- **Learn Mode:** When recording interactions with the application, WinRunner will capture new objects.
- **Edit Mode:** Allows you to manually modify an object's properties (e.g., change the name or attributes of an object).

Expected Output

- A small dialog or dropdown will appear displaying the available GUI Map Modes. Depending on the task at hand, select the appropriate mode.

Step 4: Select a Mode Based on Your Task

Goal: Choose the correct mode that fits the task you're performing.

How it Works

1 Learn Mode

- Used when you're recording interactions with new objects.
- WinRunner will automatically add objects to the GUI Map.
- This is ideal for adding objects during the test script recording phase.

© NIMI NOT TO BE REPUBLISHED

2 **Edit Mode**

- Used when you want to modify existing objects in the GUI Map.
- You can change the object name, type, properties, or any other aspect of the object.
- This mode is helpful if an object's properties change, and you need to update the GUI Map to reflect those changes.

3 **Playback Mode**

- Used during test execution when you need to use the objects already stored in the GUI Map for playback and validation.
- This mode executes the test scripts that use the stored objects.

4 **Descriptive Programming Mode**

- Used for dynamic object recognition, where objects are not referenced by their GUI Map entry but by their properties in the code.
- This mode is typically used in situations where object properties change at runtime or when dealing with objects that are dynamically generated.

Expected Output

- **Mode Selection Dialog:** You'll be able to select one of the modes, such as:
  - **Learn Mode** (for recording new objects)
  - **Edit Mode** (for modifying existing objects)
  - **Playback Mode** (for executing tests)
  - **Descriptive Programming Mode** (for dynamic object handling)

Step 5: Switch to the Desired Mode

Goal: After selecting the appropriate mode, WinRunner will switch to that mode and allow you to interact accordingly.

How it Works

1 Once you've selected the desired mode, click OK or Apply (depending on the dialog).

2 WinRunner will now enter the selected mode, and the interface will adapt based on the mode chosen:

- **Learn Mode:** Ready to capture new objects and add them to the GUI Map.
- **Edit Mode:** Allows you to manually adjust properties of the existing objects.
- **Playback Mode:** Ready to execute the test using the objects in the GUI Map.
- **Descriptive Programming Mode:** You can use scripting to dynamically interact with objects.

Expected Output

- WinRunner's interface will update based on the selected mode.
  - **Learn Mode:** You can begin recording interactions.
  - **Edit Mode:** You can see the list of objects and edit their properties.
  - **Playback Mode:** You can execute the test.
  - **Descriptive Programming Mode:** You can write and execute scripts that reference objects by their dynamic properties.

Step 6: Perform Actions in the Selected Mode

Goal: Perform your desired task depending on the selected GUI Map Mode.

How it Works

- **In Learn Mode**
  - Begin interacting with the application to add objects to the GUI Map.
  - WinRunner will automatically detect and capture the objects.
- **In Edit Mode**
  - You can open the GUI Map and edit the properties of an object.
  - This includes changing an object's name, type, window, or even other attributes.
- **In Playback Mode**
  - Run your test scripts, and WinRunner will interact with the objects as per the recorded actions.
- **In Descriptive Programming Mode**
  - You will write code to reference objects dynamically using their properties (e.g., name, type, etc.) without explicitly adding them to the GUI Map.

Expected Output

- **In Learn Mode:** New objects are added to the GUI Map as you interact with the application.
- **In Edit Mode:** Objects in the GUI Map are modified according to your changes.
- **In Playback Mode:** The test runs using the objects from the GUI Map.
- **In Descriptive Programming Mode:** Objects are identified based on their properties dynamically.

Step 7: Save Changes and Exit the GUI Map

Goal: Save your work and exit the GUI Map after completing your task.

How it Works

1 If you have made changes to the GUI Map, save those changes before exiting by selecting File > Save.

2 Exit the GUI Map when you're finished.

3 If you were in Learn Mode, the objects added to the GUI Map are automatically saved.

4 If you were in Edit Mode, save the changes before closing.

Expected Output

- If new objects were added or existing ones modified, the **GUI Map** will now reflect those changes.
- You can now exit the GUI Map and proceed with running your test scripts or working with descriptive programming.

© NIMI NOT TO BE REPUBLISHED

Using the rapid test script wizard

Objective: At the end of this exercise you shall be able to
- **use rapid test script wizard to automate test creation.**

Requirements	
Tool/Instruments	
• A working PC	- as reqd.

PROCEDURE

The **Rapid Test script wizard** is a tool used to quickly create and automate test scripts for testing software applications, typically using platforms like **TestComplete** or similar automation tools. The wizard simplifies the process by guiding the user through the creation of a test script, covering things like test objectives, test objects, and the steps to be executed.

Here's a **step-by-step guide** for using a typical Rapid Test Script Wizard (such as TestComplete's or another tool). While the exact details may vary depending on the tool you are using, the general process is similar.

Step 1: Start the Rapid Test Script Wizard

1 Open your test automation tool (e.g., **TestComplete**).

2 In the **Project Explorer** or menu bar, find the **Rapid Test Script Wizard** option.

3 Select **Create New Test** or **Create Script** to launch the wizard.

Output

The wizard interface is presented, typically with a "Welcome" or "Introduction" screen.

Step 2: Select the Test Type

The wizard will ask for the type of test you want to create:

1 Choose between different test types such as **functional test, regression test,** or **UI test.**

2 Specify the application or the system you want to test (e.g., a desktop application, web app, mobile app).

Output

The test type and application type are selected and displayed on the screen.

Step 3: Choose a Starting Point

You may be asked to choose how you want to start your test

1 Record Actions: This option will allow you to record your interactions with the application.

2 Use a Template: Select from predefined templates for specific use cases.

3 Write from Scratch: If you have a clear test scenario, you may choose to manually script the test.

Output

The wizard updates to reflect your choice, either opening the recording mode or providing you with a template for the script.

Step 4: Record Test Actions (if applicable)

If you choose the Record Actions option, you'll be guided through

1 **Launch the application:** The wizard will prompt you to launch the app you are testing.

2 **Perform actions:** As you interact with the app (click buttons, enter data, etc.), the tool records your actions.

3 **Stop recording:** After completing the required steps, stop the recording to finalize the actions.

Output

A recorded sequence of actions is generated in the script. You'll see the recorded steps in the wizard interface.

Step 5: Review and Edit the Test Steps

After recording the test steps:

1 You can review the generated test script.

2 Modify actions, add assertions (e.g., checking values, verifying states), or insert delays if needed.

Output

A detailed list of actions with parameters, conditions, and validation points is displayed in the wizard.

Step 6: Add Checkpoints or Assertions (Optional)

You can enhance the script by adding:

1 **Assertions:** These ensure the application behaves as expected. For example, verifying a button is visible or checking if a value is correct.

2 **Checkpoints:** These are specific verification points (like verifying content in a UI field).

© NIMI NOT TO BE REPUBLISHED

Output

Assertions and checkpoints are added to the script, which may be highlighted in the wizard interface.

Step 7: Configure Test Settings

You might be asked to configure

1 **Test Environment:** Specify the environment (e.g., OS, browser for web apps).

2 **Test Data:** Configure any test data you need, such as login credentials or product information.

3 **Run Parameters:** Adjust parameters like the number of repetitions or the run time.

Output

Test environment, data, and parameters are configured and shown in the wizard.

Step 8: Save and Run the Test

Once everything is set

1 Click Save to save the generated test script to your project.

2 You can choose to run the test immediately by selecting Run Test.

Output

- The script is saved and executed.
- If running, you'll see the results of the execution (pass/fail, logs, etc.).

Step 9: Review Test Results

After running the test, you'll get a Test Report showing:

1 **Test Execution Logs:** Details of what actions were performed and any issues encountered.

2 **Pass/Fail Status:** Whether the test passed or failed based on the defined conditions.

3 **Screenshots/Logs:** Any captured information like screenshots for failed steps.

Output

Test results are displayed in the report. It may include a pass/fail summary, detailed logs, and screenshots.

Example Output

For a **web login test**, the final output might look like this:

- **Test Script Summary**
  - Test Type: **Functional Test** (Web)
  - Test Steps:
    i Open Browser
    ii Navigate to Login Page
    iii Enter Credentials (username: admin, password: admin123)
    iv Click "Login" button
    v Verify Home Page loads
  - Assertions:
    i Verify Home Page Title is "Dashboard"
    ii Verify User Name is displayed as "Admin"
- **Test Run Result**
  - Test Passes: Yes
  - Total Execution Time: 5 seconds
  - Logs:
    i Step 1: Passed
    ii Step 2: Passed
    iii Assertion 1: Passed

Step 10: Optional - Debug or Enhance the Test

If the test fails or you want to improve the script, use the debugging features to:

1 Inspect specific steps.

2 Modify or add more actions.

3 Fine-tune assertions or actions.

Output

The script is adjusted based on your debugging, and the test is ready for another run.

© NIMI NOT TO BE REPUBLISHED

Choosing a record mode

Objective: At the end of this exercise you shall be able to
- **selecting the appropriate recording mode for efficient test automation.**

Requirements	
Tool/Instruments	
• A working PC	- as reqd.

PROCEDURE

Choosing **Record Mode** in an automated testing tool, such as **TestComplete**, allows you to easily capture user actions and translate them into a test script. The Record Mode typically enables you to interact with your application as usual, while the tool records your actions like clicks, keystrokes, and interactions with the UI. After recording, the tool generates a script with the recorded steps that you can edit, enhance, and reuse for automated testing.

Here's a **step-by-step guide** on how to choose and use **Record Mode** in such a tool

Step 1: Open Your Automation Tool (TestComplete)

1 Launch the automation tool (e.g., TestComplete).

2 Open the project where you want to add the test script or create a new project if necessary.

Output:

You will see the main dashboard or project explorer of the tool, where you can manage all your tests and scripts.

Step 2: Select the "Record" Mode

1 In the TestComplete interface, look for the option to Record a test. Typically, this is found in the "Tools" menu or toolbar.
   - Look for a Record button or option (e.g., "Start Recording").

2 Select Record to enter recording mode. You may be prompted to select the type of test (e.g., Desktop, Web, or Mobile).

Output

You will be prompted to specify your testing environment and the type of application (desktop, web, mobile) that you want to record actions for.

Step 3: Choose the Type of Application to Record

1 For Web Testing
   - Select the browser where your web application is running (Chrome, Firefox, etc.).
   - Choose the URL of the web app you want to test.

2 For Desktop or Mobile Testing
   - Select the application window or mobile emulator that you want to record actions on.

Output

You'll see a screen or window showing the application or browser where you can perform actions that will be recorded.

Step 4: Start Recording

1 After selecting the application or environment, click the "Start Recording" button.

2 The tool will now begin recording your actions in the selected application.

Output

The screen will switch to a "Recording" mode. A status bar or indicator will appear, confirming that recording is in progress.

Step 5: Perform Actions in the Application

1 Perform the actions you want to automate. For example:
   - Click buttons, enter text in input fields, select checkboxes, or navigate through menus.
   - The tool records every action (e.g., clicks, text inputs) and generates the corresponding commands in the script.

Output

The test script is updated in real-time with the actions you perform. The actions are shown in the TestComplete Log or script editor as they are being recorded.

Step 6: Stop Recording

1 After completing the desired actions, go back to the TestComplete interface.

2 Click the "Stop Recording" button to end the recording session.

Output

The tool will finish capturing your interactions and stop recording. The recorded test steps are saved and displayed in the Script Editor or Keyword Test tab.

Step 7: Review the Recorded Actions

1 Open the Test Editor to see the recorded script. The actions you took will appear as commands in the script, such as:

© NIMI NOT TO BE REPUBLISHED

- ClickButton("LoginButton")
- SetText("usernameField", "admin")
- SetText("passwordField", "password123")
- ClickButton("submitButton")

2 Review the recorded actions to ensure they match the steps you wanted to test.

Output

You'll see a sequence of test steps or code that corresponds to each action you performed. You may also notice that TestComplete adds wait times, checks, or screenshots based on the recorded steps.

Step 8: Modify the Recorded Test (Optional)

1 If needed, you can modify the recorded steps to improve the test:
   - Add assertions or checkpoints to validate UI elements (e.g., verifying that a page loads correctly).
   - Edit the recorded code for better readability or to handle specific conditions.
   - Adjust the delays or timeouts between actions.

Output

The script is now customized, and additional checks or validations are added. You can refine the test to make it more robust.

Step 9: Save the Recorded Test Script

1 Once you are satisfied with the recorded actions, click the "Save" button to save the test script.

2 You may also choose to name your script for easy reference.

Output

The recorded test is saved in your project and is now ready for execution.

Step 10: Run the Recorded Test

1 After saving the script, you can choose to run the test immediately.

2 Click the "Run" button to execute the test.

Output

The tool will run the recorded test, simulating the actions you performed in the application. The results will be displayed in a test result log, showing whether the test passed or failed.

Example of Recorded Actions (Output)

For a simple login test, the recorded actions may look like this:

Recorded Script Example (Fig 1)

Fig 1

```
// Launching the browser
Sys.Browser("chrome").ToUrl("https://example.com/login");

// Entering username
Aliases.browser.pageLogin.textboxUsername.SetText("admin");

// Entering password
Aliases.browser.pageLogin.textboxPassword.SetText("password123");

// Clicking login button
Aliases.browser.pageLogin.buttonLogin.Click();

// Verifying login success
if (Aliases.browser.pageDashboard.Exists) {
  Log.Message("Login Successful");
} else {
  Log.Error("Login Failed");
}
```

Output

- The browser opens and navigates to the login page.
- It enters credentials and clicks the login button.
- The test checks if the dashboard page exists, logging success or failure.

Step 11: Review the Test Results

After running the test, you'll see a result report:

- **Test Pass:** If all actions and validations were successful.
- **Test Fail:** If any of the actions or validations failed.

Output

The result screen will display a summary of the test's success or failure, along with detailed logs and screenshots if any step failed.

© NIMI NOT TO BE REPUBLISHED

Recording a context sensitive test

Objective: At the end of this exercise you shall be able to
- **capturing user interactions with GUI objects for test automation.**

Requirements

Tool/Instruments
- A working PC - as reqd.

PROCEDURE

Recording a context-sensitive test involves capturing the interactions of a user with an application while maintaining the relationship between UI elements and their logical identifiers. This ensures that the test can run even if UI positions change. Below is a step-by-step guide to recording a context-sensitive test.

Step-by-Step Guide to Recording a Context-Sensitive Test

Step 1: Launch the Testing Tool
- Open the test automation tool that supports context-sensitive testing (e.g., Micro Focus UFT, Selenium, TestComplete).
- Ensure the application under test (AUT) is available.

Step 2: Start Test Recording
- Click on Record in the test automation tool.
- Choose Context-Sensitive Mode (if available).
- Select the application or the part of the UI to record.

Step 3: Perform User Actions
- Perform the actions you want to test (e.g., clicking buttons, entering text, navigating menus).
- The tool records each action along with object properties like:
  - Object name
  - Type (button, text box, dropdown, etc.)
  - Associated properties (ID, Name, XPath, etc.)

Example of recorded script (UFT/QTP format) (Fig 1)

Fig 1

```
Browser("Google").Page("Google").WebEdit("q").Set "Context-sensitive test"
Browser("Google").Page("Google").WebButton("Google Search").Click
```

Step 4: Validate Recorded Steps
- Stop the recording.
- Review the recorded steps to ensure they correctly map to UI elements.

Step 5: Insert Verification Points (Assertions)
- Add checkpoints or assertions to verify expected behavior.
- Example (Selenium with Java) (Fig 2)

Fig 2

```
WebElement searchBox = driver.findElement(By.name("q"));
assert searchBox.isDisplayed();
```

Step 6: Parameterize the Test
- Replace hardcoded values with variables to enhance reusability.
- Example (TestComplete JavaScript) (Fig 3)

Fig 3

```
var searchValue = "Automation Testing";
Aliases.browser.pageGoogle.textboxSearch.SetText(searchValue);
```

Step 7: Save and Run the Test
- Save the recorded test script.
- Execute the test and observe results.

Step 8: Analyze Output & Debug Failures
- Review the test log for errors.
- Debug issues if necessary (e.g., object recognition failures).
- Modify object properties or use dynamic locators if needed.

Expected Output
1. The test tool logs interactions.
2. The test executes successfully, interacting with the UI as expected.
3. If errors occur, they are displayed in the test log.
4. Assertions verify expected UI behavior.

© NIMI NOT TO BE REPUBLISHED

Understanding the text script

Objective: At the end of this exercise you shall be able to
- **analyzing recorded actions for test script generation.**

Requirements

Tool/Instruments
- A working PC - as reqd.

PROCEDURE

It looks like you're asking for an understanding of the **text script** used for recording context-sensitive tests in the example above. Let me break down each part of the script step by step:

Example Test Script

We will use a Selenium WebDriver script in Java for better understanding. The script automates a Google search and verifies the result.

Test Scenario

1 Open Google in a web browser.

2 Enter "context-sensitive test" in the search box.

3 Click on the **Google Search** button.

4 Verify that the results page contains relevant search results.

Step-by-Step Breakdown of the Script

Step 1: Import Required Libraries (Fig 1)

Fig 1

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
```

Explanation

- These are Selenium libraries needed to control the browser and locate elements.
- WebDriver is used to interact with the browser.
- By is used to locate UI elements.

Step 2: Initialize WebDriver and Open Google (Fig 2)

Fig 2

```
public class ContextSensitiveTest {
    public static void main(String[] args) {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        WebDriver driver = new ChromeDriver();
        driver.get("https://www.google.com");
    }
}
```

Expected Output in Browser

- A new Chrome browser opens.
- Google's homepage is displayed.

Step 3: Locate the Search Box and Enter Text (Fig 3)

Fig 3

```
WebElement searchBox = driver.findElement(By.name("q"));
searchBox.sendKeys("context-sensitive test");
```

Explanation

- findElement(By.name("q")) finds the Google search box using its **name attribute** (q).
- sendKeys("context-sensitive test") enters the search text.

Expected Output in Browser

- "context-sensitive test" is typed into the Google search box.

Step 4: Click the Search Button (Fig)

Fig 4

```
searchBox.submit();
```

Explanation

- submit() simulates pressing the "Enter" key to start the search.

© NIMI NOT TO BE REPUBLISHED

Expected Output in Browser

- Google search results appear for "context-sensitive test".

Step 5: Verify the Search Results (Fig 5)

Fig 5

```
WebElement firstResult = driver.findElement(By.cssSelector("h3"));
System.out.println("First result: " + firstResult.getText());
```

Explanation

- findElement(By.cssSelector("h3")) finds the first search result title.
- getText() retrieves and prints the result title.

Expected Output in Console (Fig 6)

Fig 6

```
First result: Context-Sensitive Testing Explained
```

Step 6: Close the Browser (Fig 7)

Fig 7

```
driver.quit();
```

Explanation

- quit() closes the browser session.

Expected Output

- The browser closes successfully.

Final Script (Complete Code) (Fig 8)

Fig 8

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class ContextSensitiveTest {
    public static void main(String[] args) {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        WebDriver driver = new ChromeDriver();

        // Step 1: Open Google
        driver.get("https://www.google.com");

        // Step 2: Locate search box and enter text
        WebElement searchBox = driver.findElement(By.name("q"));
        searchBox.sendKeys("context-sensitive test");

        // Step 3: Submit search query
        searchBox.submit();

        // Step 4: Wait for results and verify the first result
        try {
            Thread.sleep(2000); // Wait for results to load
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        WebElement firstResult = driver.findElement(By.cssSelector("h3"));
        System.out.println("First result: " + firstResult.getText());

        // Step 5: Close browser
        driver.quit();
    }
}
```

© NIMI NOT TO BE REPUBLISHED

Final Expected Output

1 Browser Output

- Google search opens.
- "context-sensitive test" is typed in.
- Search executes, and results appear.

2 Console Output (Fig 9)

Fig 9

```
First result: Context-Sensitive Testing Explained
```

Recording in analogy mode

Objective: At the end of this exercise you shall be able to
- **creating tests by recording actions and analogies.**

Requirements

Tool/Instruments
- A working PC - as reqd.

PROCEDURE

Recording in Analogy Mode - Step by Step Guide with Output

Analogy Mode in test recording captures user interactions **relative to the screen layout**, using **coordinates or pixel positions** instead of logical object recognition (context-sensitive mode). This method is useful when dealing with applications where UI elements lack unique identifiers, such as graphical apps or legacy systems.

Step-by-Step Process for Recording in Analogy Mode

Step 1: Launch the Test Automation Tool
- Open your testing tool (e.g., **Micro Focus UFT, TestComplete, AutoIT, etc.**).
- Select **Analogy Mode** or **Low-Level Recording Mode** (depending on the tool).
- Start a **new test script**.

Expected Output
- The recording mode is enabled.

Step 2: Start Recording in Analogy Mode
- Click Record to start capturing actions.
- The tool will log exact mouse movements, clicks, and keystrokes as pixel-based coordinates.

Expected Output
- The tool begins recording all screen interactions in terms of X, Y positions.

Step 3: Perform Actions on the Application
- Click buttons, type text, and navigate the UI.
- The tool captures actions with absolute coordinates.

Example of Recorded Script (TestComplete - JavaScript) (Fig 1)

Fig 1

```
// Open application
Sys.Process("Notepad").Window("Notepad").Click(100, 200);

// Type text
Sys.Process("Notepad").Window("Notepad").Keys("Hello, Analogy Mode!");

// Click Save button (assuming its position)
Sys.Process("Notepad").Window("Notepad").Click(400, 500);
```

Expected Output
- The cursor moves to position (100, 200) and clicks.
- Text is typed into Notepad.
- The save button is clicked at position (400, 500).

Step 4: Stop Recording and Review the Script
- Stop the recording.
- The script contains coordinate-based steps instead of logical object identifiers.

Expected Output
- A script with mouse clicks and keyboard actions based on screen positions.

Step 5: Play Back the Test
- Run the recorded test.
- The tool replicates the exact movements and keystrokes based on recorded positions.

Expected Output
- Actions replay exactly as recorded.
- Test may fail if the UI layout has changed (since positions are hardcoded).

© NIMI NOT TO BE REPUBLISHED

Comparison: Context-Sensitive vs. Analogy Mode

Feature	Context-Sensitive Mode	Analogy Mode
Object Recognition	Identifies elements by name, ID, or XPath	Uses X, Y coordinates
UI Changes Handling	Adapts to UI changes	Fails if UI layout changes
Best for	Web and modern apps with proper UI elements	Graphical apps, legacy systems
Recorded Steps	Logical identifiers	Absolute positions

© NIMI
NOT TO BE REPUBLISHED

Running the test

Objective: At the end of this exercise you shall be able to
- **executing recorded tests to validate application behavior.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Running the Test in Analogy Mode - Step by Step Guide with Expected Output

After recording the test in **Analogy Mode**, the next step is to execute the test and observe the output.

Step-by-Step Execution of an Analogy Mode Test

Step 1: Open the Test Automation Tool

- Open the tool used for recording **(TestComplete, UFT, AutoIT, etc.)**
- Load the **recorded test script.**

Expected Output

- The test script is loaded in the tool, ready for execution.

Step 2: Launch the Application Under Test (AUT)

- Ensure the application (e.g., Notepad, Calculator, a GUI app) is open and in the same position as when recorded.
- If necessary, the script may start the application automatically.

Example Script (TestComplete - JavaScript) (Fig 1)

Fig 1

```
// Step 1: Launch Notepad
Sys.Process("notepad").Window("Notepad").Activate();
```

Expected Output

- **Notepad application opens** (if not already open).

Step 3: Execute the Recorded Steps

- Run the test by clicking "Run" in the automation tool.

Example Steps (Recorded in Analogy Mode) (Fig 2)

Fig 2

```
// Step 2: Click on a specific location (100, 200)
Sys.Process("notepad").Window("Notepad").Click(100, 200);

// Step 3: Type text
Sys.Process("notepad").Window("Notepad").Keys("Hello, Analogy Mode!");

// Step 4: Click Save button at coordinates (400, 500)
Sys.Process("notepad").Window("Notepad").Click(400, 500);
```

Expected Output

- The cursor moves to (100, 200) and clicks on the text area.
- "Hello, Analogy Mode!" is typed into Notepad.
- The cursor moves to (400, 500) and clicks the Save button.

Step 4: Validate Test Execution

- Check the execution log for errors or warnings.
- If the test fails, ensure that:
  - The screen resolution is the same.
  - The window is in the correct position.
  - The UI layout hasn't changed.

© NIMI NOT TO BE REPUBLISHED

Example Console Log (Success Case) (Fig 3)

Fig 3

```
Test Execution Started...
Opening Notepad...
Clicking at (100, 200)...
Typing text: "Hello, Analogy Mode!"...
Clicking Save button at (400, 500)...
Test Execution Completed Successfully!
```

Example Console Log (Failure Case) (Fig 4)

Fig 4

```
Test Execution Started...
Opening Notepad...
Error: Unable to find the window at (100, 200).
Test Failed: UI layout might have changed.
```

Step 5: Close the Application and Finish Execution

- If the test script does not include a close command, manually close the application.
- Log the results.

Example Script to Close Notepad (Fig 5)

Fig 5

```
// Step 5: Close Notepad
Sys.Process("notepad").Window("Notepad").Close();
```

Expected Output

- Notepad closes after test execution.

© NIMI
NOT TO BE REPUBLISHED

Analysing test results

Objective: At the end of this exercise you shall be able to
- **evaluating test execution outcomes and identifying failures.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Analyzing Test Results Step by Step with Expected Output

Once the test has been executed, analyzing the results is crucial to determine whether it passed or failed and to diagnose potential issues.

Step-by-Step Process for Analyzing Test Results

Step 1: Open the Test Execution Report

- After the test run, open the test report/log in your automation tool.
- Common tools and their result formats:
  - **UFT (Micro Focus):** HTML Test Report
  - **TestComplete:** Log Window (structured test steps)
  - **Selenium with TestNG/JUnit:** Console Output + XML/HTML Reports
  - **AutoIT:** Log files

Expected Output

- A report showing pass/fail status of each test step.

Step 2: Check Overall Test Status

- Look at the final test execution summary to determine if the test passed or failed.

Example: Test Report Summary (Successful Execution) (Fig 1)

Fig 1

```
Test Execution Report - Analogy Mode Test
----------------------------------------
Test Case: Verify Notepad Typing and Save
Execution Time: 12.5s
Status: ✅ PASSED
Total Steps: 4
Steps Passed: 4
Steps Failed: 0
```

Example: Test Report Summary (Failed Execution) (Fig 2)

Fig 2

```
Test Execution Report - Analogy Mode Test
----------------------------------------
Test Case: Verify Notepad Typing and Save
Execution Time: 10.2s
Status: ❌ FAILED
Total Steps: 4
Steps Passed: 3
Steps Failed: 1
Failed Step: Click Save Button at (400,500) - Button not found
```

Step 3: Examine Detailed Step-by-Step Execution Log

- Open the detailed test execution log to identify failures or unexpected behavior.
- Look for screenshots (if available) or error messages.

© NIMI NOT TO BE REPUBLISHED

Example: Successful Step-by-Step Execution Log (Fig 3)

Fig 3

```
1. Open Notepad - ✅ SUCCESS
2. Click at (100, 200) - ✅ SUCCESS
3. Type 'Hello, Analogy Mode!' - ✅ SUCCESS
4. Click Save at (400, 500) - ✅ SUCCESS
```

Example: Failed Step in Execution Log (Fig 4)

Fig 4

```
1. Open Notepad - ✅ SUCCESS
2. Click at (100, 200) - ✅ SUCCESS
3. Type 'Hello, Analogy Mode!' - ✅ SUCCESS
4. Click Save at (400, 500) - ❌ FAILED (Button not found)
   Error: UI element not present at expected location.
```

Step 4: Identify Root Cause of Failures

If any step failed, analyze potential reasons:

- **UI Layout Changed?** - The element's position may have shifted.
- **Application Not Loaded?** - The app may not have opened properly.
- **Wrong Window Focus?** - Another window may have interfered.
- **Timing Issue?** - The test may have clicked too soon.

Example: Console Error Message (UI Layout Change Issue) (Fig 5)

Fig 5

```
Error: Cannot click at (400, 500) - No button found at this location.
Possible Causes:
1. UI layout changed.
2. Application not fully loaded before action.
3. Incorrect screen resolution.
```

Example: Fix - Adding a Delay for UI Load (Fig 6)

Fig 6

```
// Adding delay to wait for UI elements to load
aqUtils.Delay(2000);
Sys.Process("notepad").Window("Notepad").Click(400, 500);
```

Step 5: Verify Screenshots and Logs

- Many tools provide screenshots at failure points.
- Check whether the UI element was present at the expected location.

Example: Screenshot Analysis

- The Save button was originally at (400, 500) but moved to (420, 520) due to a UI update.

Fix: Update test script with new coordinates.

Step 6: Re-run the Test After Fixing Issues

- Modify the test script based on the analysis.
- Re-run the test to verify if the issue is resolved.

Expected Output (After Fix) (Fig 7)

Fig 7

```
Test Execution Report - Analogy Mode Test
-----------------------------------------
Test Case: Verify Notepad Typing and Save
Execution Time: 11.8s
Status: ✅ PASSED
```

© NIMI NOT TO BE REPUBLISHED

Recording tips

Objective: At the end of this exercise you shall be able to
- **improving recording efficiency and test script quality.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Recording Tips for Test Automation – Step by Step with Expected Output

Recording a test properly ensures **stability, reliability, and reusability**. Below are step-by-step **best practices and tips** to follow when recording a test.

Step 1: Choose the Right Recording Mode

Most test automation tools offer different recording modes:

- **Context-Sensitive Mode** – Identifies elements by **name, ID, XPath** (recommended for stable apps).
- **Analogy Mode (Low-Level Recording)** – Captures **exact screen coordinates** (use for graphical or legacy apps).
- **Hybrid Mode** – Uses a combination of both (useful for apps with mixed UI types).

Example: Selecting Recording Mode in UFT (Fig 1)

Fig 1

```
Tools → Options → Recording → Choose "Context-Sensitive Mode"
```

Expected Output

- The tool is set to record element properties instead of raw mouse clicks.

Step 2: Prepare the Application Under Test (AUT)

- **Ensure AUT is in a default state** (e.g., login page or home screen).
- **Maximize the window** to prevent resolution issues.
- **Close unnecessary applications** to avoid interference.

Example Setup Log (TestComplete) (Fig 2)

Fig 2

```
Application Launched: Notepad.exe
Window Maximized: TRUE
Background Apps Closed: Chrome, Slack
```

Expected Output

- The application is ready for recording with minimal interference.

Step 3: Start Recording & Perform Actions Slowly

- Click the Record button in your automation tool.
- Perform actions slowly and naturally to ensure clear element recognition.
- Avoid unnecessary movements like dragging the mouse randomly.

Example: Recording Start in Selenium IDE (Fig 3)

Fig 3

```
Click "Record" → Perform actions → Click "Stop"
```

Expected Output

- A structured test script is generated without extra/ unwanted steps.

Step 4: Use Unique Identifiers Instead of Coordinates

- Avoid hardcoded X, Y positions unless necessary.
- Prefer XPath, ID, Name, Class, or CSS Selectors to locate elements.

Bad Example (Hardcoded Coordinates - Fragile) (Fig 4)

Fig 4

```
driver.findElement(By.xpath("//div[4]/button")).click();
```

© NIMI NOT TO BE REPUBLISHED

Good Example (Using Unique Identifiers - Stable)(Fig 5)

Fig 5

```
driver.findElement(By.id("submitBtn")).click();
```

Expected Output

- The test remains robust even if UI layout changes.

Step 5: Add Wait Commands to Handle Timing Issues

- Use implicit or explicit waits to handle slow-loading elements.
- Avoid using Thread.sleep() as it slows execution unnecessarily.

Example: Adding Explicit Wait (Selenium - Java) (Fig 6)

Fig 6

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("loginButton")));
element.click();
```

Expected Output

- The test waits dynamically until the element appears, preventing failures.

Step 6: Use Assertions for Validation

- Always include assertions to check expected vs. actual results.
- Helps in debugging failures quickly.

Example: Assertion for Text Validation (JUnit - Java) (Fig 7)

Fig 7

```
String expectedTitle = "Dashboard - My App";
String actualTitle = driver.getTitle();
Assert.assertEquals(expectedTitle, actualTitle);
```

Expected Output

- The test fails if the title doesn't match, making debugging easier.

Step 7: Stop Recording & Review the Script

- Click Stop Recording once all actions are captured.
- Check the generated script and remove unnecessary steps.
- Rename recorded objects for better readability.

Example: Renaming an Object in TestComplete (Fig 8)

Fig 8

```
Old: Aliases.Process.Button1
New: Aliases.Process.LoginButton
```

Expected Output

- The script is cleaner and easier to maintain.

Step 8: Save & Execute the Recorded Test

- Save the test script.
- Execute it at least twice to verify stability.

Example Execution Output (Console Log - Selenium) (Fig 9)

Fig 10

```
Test Execution Started...
Step 1: Open URL - PASSED
Step 2: Enter Username - PASSED
Step 3: Click Login - PASSED
Step 4: Verify Dashboard - PASSED
Test Execution Completed - SUCCESS
```

© NIMI NOT TO BE REPUBLISHED

Expected Output

- The test runs successfully without failures.

Final Summary of Best Practices

Step	Best Practice	Expected Outcome
1. Select Recording Mode	Use **Context-Sensitive Mode** for stable tests.	Elements are recorded using **logical identifiers.**
2. Prepare AUT	Start from a **default state** and **maximize** the window.	No **background interference** during recording.
3. Record Actions Slowly	Perform steps **naturally** and **avoid unnecessary clicks.**	Cleaner script with **only relevant steps.**
4. Use Unique Identifiers	Avoid **hardcoded coordinates**, use **IDs/XPath/CSS.**	Test remains **stable** even if UI changes.
5. Add Waits	Use **explicit waits** instead of `Thread.sleep()`.	Prevents **flaky tests due to timing issues.**
6. Use Assertions	Validate expected vs. actual results.	Errors are **detected early** with clear messages.
7. Review Script	Remove extra steps and rename objects.	Improves **readability** and **maintainability.**
8. Save & Execute	Run the test **multiple times.**	Ensures the test is **stable and repeatable.**

© NIMI NOT TO BE REPUBLISHED

When should you synchronize?

Objective: At the end of this exercise you shall be able to
- **evaluating test execution outcomes and identifying failures.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

When Should You Synchronize in Test Automation? (Step by Step with Expected Output)

What is Synchronization?

Synchronization ensures that test scripts **wait for an application to respond** before executing the next step. This prevents **flaky tests** caused by slow UI elements, network latency, or background processing.

Step-by-Step Guide on When to Synchronize

Step 1: Synchronize Before Interacting with UI Elements

- Some elements may take time to load or become visible after an action.
- If the script tries to interact too early, it may fail.

Example (Selenium - Java) (Fig 1)

Fig 1

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
WebElement loginButton = wait.until(ExpectedConditions.elementToBeClickable(By.id("loginBtn")));
loginButton.click();
```

Expected Output

- The script waits until the login button is clickable before clicking it.

Step 2: Synchronize After Page Navigation or Refresh

- When navigating between pages, elements may take time to load.
- Ensure that the new page is fully loaded before performing further actions.

Example (Selenium - Java) (Fig 2)

Fig 2

```
driver.get("https://example.com/dashboard");
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
wait.until(ExpectedConditions.titleContains("Dashboard"));
```

Expected Output

- The test waits until the Dashboard page title appears, ensuring it's loaded.

Step 3: Synchronize When Waiting for Dynamic Content

- Some elements load dynamically after the page is displayed (AJAX calls, pop-ups, etc.).
- Synchronization is necessary to wait for new elements to appear.

© NIMI NOT TO BE REPUBLISHED

Example (TestComplete - JavaScript) (Fig 3)

Fig 3

```
var panel = Sys.Browser("chrome").Page("https://example.com").WaitElement("//div[@id='dynamicPanel']", 5000);
if (panel.Exists) {
    panel.Click();
}
```

Expected Output

- The test waits for up to 5 seconds until dynamicPanel appears before clicking it.

Step 4: Synchronize When Handling Animations or Loading Spinners

- Some applications use loading indicators (spinners, progress bars).
- The test should wait until the loader disappears before proceeding.

Example (Selenium - Java) (Fig 4)

Fig 4

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
wait.until(ExpectedConditions.invisibilityOfElementLocated(By.id("loadingSpinner")));
```

Expected Output

- The script waits until the spinner disappears before executing the next step.

Step 5: Synchronize When Verifying Asynchronous Actions

- Actions like form submission or data processing may take time.
- Ensure that the success message or updated content appears before proceeding.

Example (Cypress - JavaScript) (Fig 5)

Fig 5

```
cy.get("#submitButton").click();
cy.get("#successMessage").should("be.visible");
```

Expected Output

- The test waits until the success message appears after submission.

Step 6: Synchronize When Dealing with Multi-Window or iFrames

- If switching between windows or frames, ensure the target frame is loaded first.

Example (Selenium - Java) (Fig 6)

Fig 6

```
driver.switchTo().frame("frameID");
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
wait.until(ExpectedConditions.presenceOfElementLocated(By.id("frameElement")));
```

Expected Output

- The script waits until elements inside the iFrame are available before interacting.

Step 7: Synchronize Before Executing JavaScript Events

- Some JavaScript events (modals, alerts) may take time to trigger.
- Ensure they appear before interacting.

© NIMI NOT TO BE REPUBLISHED

Example (Selenium - Java) (Fig 7)

Fig 7

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(5));
wait.until(ExpectedConditions.alertIsPresent());
Alert alert = driver.switchTo().alert();
alert.accept();
```

Expected Output

- The test waits until an alert appears before accepting it.

Final Summary: When to Synchronize?

Scenario	Why Synchronize?	Solution
Before clicking elements	Ensure element is visible & clickable	`wait.until(ExpectedConditions.elementToBeClickable())`
After navigation	Ensure the page is fully loaded	`wait.until(ExpectedConditions.titleContains())`
For dynamic content (AJAX, pop-ups)	Wait for the element to appear	`wait.until(ExpectedConditions.presenceOfElementLocated())`
During animations/loading spinners	Wait for UI elements to stabilize	`wait.until(ExpectedConditions.invisibilityOfElementLocated())`
After form submission	Wait for confirmation message	`wait.until(ExpectedConditions.visibilityOfElementLocated())`
When switching iFrames/windows	Ensure new context is ready	`wait.until(ExpectedConditions.presenceOfElementLocated())`
Before handling JavaScript alerts	Ensure alert box appears	`wait.until(ExpectedConditions.alertIsPresent())`

© NIMI NOT TO BE REPUBLISHED

Creating a test

Objective: At the end of this exercise you shall be able to
- **create tests to manage timing and dependencies.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Creating a Test Step by Step with Output

Overview

A test case consists of **preconditions, test steps, expected results, and assertions** to verify an application's functionality. Below is a **step-by-step guide** to creating a test, along with expected outputs.

Step 1: Define Test Case Objectives

Before writing a test, determine:

- **What feature or function will be tested?**
- **What is the expected behavior?**

Example Test Objective:

Verify that a user can successfully log in to a website using valid credentials.

Step 2: Identify Preconditions

- Ensure the application under test (AUT) is installed/ running.
- Ensure test data (username/password) is available.

Example Preconditions (Fig 1)

Fig 1

```
1. The application must be accessible at https://example.com/login.
2. The user must have a valid username and password.
3. The browser should be open and maximized.
```

Expected Output

- The browser launches and navigates to the login page.

Step 3: Write Test Steps

A structured test has clear, step-by-step instructions.

Example Test Steps for Login Functionality (Fig 2)

Fig 2

```
1. Open the browser and navigate to https://example.com/login.
2. Enter a valid username in the username field.
3. Enter the correct password in the password field.
4. Click the "Login" button.
5. Verify that the user is redirected to the dashboard page.
```

Expected Output

- The login page loads.
- The credentials are entered.
- Clicking the Login button redirects the user to the Dashboard.

Step 4: Write the Automated Test Script

Here's how the above steps translate into an automated test using **Selenium (Java).** (Fig 3)

© NIMI NOT TO BE REPUBLISHED

Fig 3

```java
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import java.time.Duration;

public class LoginTest {
    public static void main(String[] args) {
        // Step 1: Setup WebDriver
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        WebDriver driver = new ChromeDriver();

        // Step 2: Open browser and navigate to login page
        driver.get("https://example.com/login");
        driver.manage().window().maximize();

        // Step 3: Enter username and password
        driver.findElement(By.id("username")).sendKeys("testuser");
        driver.findElement(By.id("password")).sendKeys("password123");

        // Step 4: Click Login button
        driver.findElement(By.id("loginButton")).click();

        // Step 5: Verify successful login by checking page title
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(5));
        String expectedTitle = "Dashboard - Example";
        String actualTitle = driver.getTitle();
        Assert.assertEquals(actualTitle, expectedTitle, "Login Failed!");

        // Step 6: Close the browser
        driver.quit();
    }
}
```

© NIMI NOT TO BE REPUBLISHED

Expected Output (Console Log) (Fig 4)

Fig 4

```
Launching Chrome browser...
Navigating to https://example.com/login...
Entering username: testuser
Entering password: ********
Clicking the Login button...
Verifying dashboard title...
Test PASSED: User successfully logged in!
Closing browser...
```

Step 5: Add Assertions to Validate the Results

Assertions help ensure expected vs. actual results match.

Example Assertion for Successful Login (Fig 5)

Fig 5

```
Assert.assertEquals(actualTitle, expectedTitle, "Login Failed!");
```

Expected Output

- If login succeeds. (Fig 6)

Fig 6

```
Test PASSED: User successfully logged in!
```

- If login fails. (Fig 7)

Fig 7

```
Test FAILED: Login Failed! Expected: 'Dashboard - Example', but got: 'Login Page'
```

Step 6: Execute the Test and Analyze the Output

Run the test script using a testing framework like JUnit or TestNG.

Example TestNG Execution Output (Fig 8)

Fig 8

```
[TestNG] Running:
Tests started...
Launching browser...
Navigating to login page...
Entering credentials...
Clicking login...
Verifying successful login...
PASSED: LoginTest
```

Expected Output

- **If successful:** User is redirected to the dashboard.
- **If failed:** The script provides an error message (e.g., incorrect credentials, element not found).

Step 7: Generate a Test Report

Most automation tools generate a test report for analysis.

Example TestNG Report Summary (Fig 9)

Fig 9

```
Test Suite: Login Tests
--------------------------------
Test Case: Verify User Login
Status: ✅ PASSED
Execution Time: 5.2s
Assertions: 1 Passed, 0 Failed
```

© NIMI NOT TO BE REPUBLISHED

Changing the synchronization setting

Objective: At the end of this exercise you shall be able to
- **adjust synchronization settings to manage test timing.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

What is Synchronization?

Synchronization ensures that **test execution waits for the application's response** before proceeding to the next step. This helps prevent **flaky tests** caused by timing issues, slow page loads, or dynamic elements.

Step-by-Step Guide to Changing Synchronization Settings

Step 1: Open the Test Automation Tool

- Launch your automation testing tool (e.g., Selenium, UFT, TestComplete, Cypress).
- Navigate to settings/preferences to adjust synchronization.

Expected Output

- The automation tool opens, and settings are accessible.

Step 2: Adjust Implicit Wait (Global Synchronization)

Implicit waits tell the test script to wait for a certain amount of time **before throwing an error** if an element is not found.

Example (Selenium - Java) (Fig 1)

Fig 1

```
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
```

Example (UFT - VBScript) (Fig 2)

Fig 2

```
Setting("DefaultTimeout") = 10000  '10 seconds
```

Expected Output

- The test script waits for elements to appear before throwing an error.

Step 3: Implement Explicit Wait (Dynamic Synchronization)

Explicit waits allow tests to wait only when necessary.

Example (Selenium - Java) (Fig 3)

Fig 3

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
WebElement loginButton = wait.until(ExpectedConditions.elementToBeClickable(By.id("loginBtn")));
loginButton.click();
```

Example (TestComplete - JavaScript) (Fig 4)

Fig 4

```
var loginButton = Sys.Browser("chrome").Page("https://example.com").WaitElement("#loginBtn", 10000);
loginButton.Click();
```

© NIMI NOT TO BE REPUBLISHED

Expected Output

- The test dynamically waits until the element is ready, preventing failures.

Step 4: Adjust Page Load Timeout

This setting ensures the test waits for the entire page to load before proceeding.

Example (Selenium - Java) (Fig 5)

Fig 5

```
driver.manage().timeouts().pageLoadTimeout(Duration.ofSeconds(20));
```

Example (UFT - VBScript) (Fig 6)

Fig 6

```
Browser("MyBrowser").Page("MyPage").Sync
```

Expected Output

- The test script waits for the page to load completely before interacting with elements.

Step 5: Handle Loading Spinners or AJAX Calls

Sometimes, applications use loading indicators that should disappear before continuing.

Example (Selenium - Java) (Fig 7)

Fig 7

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
wait.until(ExpectedConditions.invisibilityOfElementLocated(By.id("loadingSpinner")));
```

Expected Output

- The test waits for the spinner to disappear, preventing premature clicks.

Step 6: Save and Apply Synchronization Settings

- Save the updated settings and apply them to the test script.
- Run the test to verify synchronization settings.

Expected Output

- The test executes smoothly without failures due to timing issues.

© NIMI NOT TO BE REPUBLISHED

Identifying a synchronization problem

Objective: At the end of this exercise you shall be able to
- **identify synchronization issues affecting test reliability.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

What is a Synchronization Problem?

A synchronization issue occurs when a test script **executes steps before the application is ready**, leading to:

- Element not found errors
- Stale element exceptions
- Clicks on invisible elements
- Flaky test behavior

Step 1: Run the Test and Observe Errors

- Execute your test script and check the console/logs.
- Look for **timeouts, missing elements, or stale references.**

Example Test Case:

- Open the login page
- Enter username & password
- Click the Login button
- Verify redirection to the Dashboard

Example Test Script (Selenium - Java) (Fig 1)

Fig 1

```
driver.get("https://example.com/login");
driver.findElement(By.id("username")).sendKeys("testuser");
driver.findElement(By.id("password")).sendKeys("password123");
driver.findElement(By.id("loginButton")).click();
WebElement dashboard = driver.findElement(By.id("dashboard"));
System.out.println("Test Passed: User logged in!");
```

Possible Console Output (Failure) (Fig 2)

Fig 2

```
Exception in thread "main" org.openqa.selenium.NoSuchElementException:
Unable to locate element: {"method":"id","selector":"dashboard"}
```

What Happened?

- The script tried to find the dashboard too quickly before the page loaded.

© NIMI NOT TO BE REPUBLISHED

Step 2: Analyze the Type of Synchronization Issue

Issue	Error Message	Cause
Element Not Found	`NoSuchElementException`	Element hasn't appeared yet
Stale Element Exception	`StaleElementReferenceException`	Page refreshed and element reference changed
Click on Invisible Element	`ElementNotInteractableException`	Element exists but is not clickable
Timeout Exception	`TimeoutException`	Page/element took too long to load

Step 3: Debug Using Explicit Waits

- Replace direct element interactions with explicit waits.

Updated Code with Synchronization Fix (Fig 3)

Fig 3

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
WebElement dashboard = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("dashboard")));
System.out.println("Test Passed: User logged in!");
```

Expected Console Output (Success) (Fig 4)

Fig 4

```
Test Passed: User logged in!
```

Step 4: Add Page Load Timeout

If the test fails because the page loads slowly, add a page load timeout.

Example Fix (Fig 5)

Fig 5

```
driver.manage().timeouts().pageLoadTimeout(Duration.ofSeconds(15));
```

Expected Output

- The test waits for up to 15 seconds for the page to load before proceeding.

Step 5: Handle Loading Spinners

If the app shows a loading spinner, wait until it disappears.

Example Fix (Fig 6)

Fig 6

```
wait.until(ExpectedConditions.invisibilityOfElementLocated(By.id("loadingSpinner")));
```

Expected Output

- The script waits for the spinner to disappear, preventing premature failures.

Step 6: Execute the Test Again & Validate the Fix

- Run the script with synchronization improvements.
- If errors persist, tweak the wait times or conditions.

Final Expected Console Output (Success) (Fig 7)

Fig 7

```
Navigating to login page...
Entering credentials...
Clicking login button...
Waiting for dashboard...
Test Passed: User logged in!
```

© NIMI NOT TO BE REPUBLISHED

Synchronizing the test

Objective: At the end of this exercise you shall be able to
- **ensure proper timing and dependencies for tests.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Synchronizing the test in **Synchronizing Tests** involves ensuring that all necessary conditions, resources, and dependencies are properly aligned before the test proceeds. This step is essential to avoid timing issues such as race conditions, flaky tests, or inconsistent results, especially when dealing with external systems, asynchronous operations, or multi-threaded environments. Here's how to synchronize tests effectively.

Key Steps for Synchronizing Tests

1 Set Explicit Waits

- Ensure that tests wait for certain conditions (e.g., elements, data, API responses) before proceeding.
- **Example in Selenium (Fig 1)**

Fig 1

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# Wait for the element with ID 'my_element' to be visible
WebDriverWait(driver, 10).until(
    EC.visibility_of_element_located((By.ID, 'my_element'))
)
```

© NIMI NOT TO BE REPUBLISHED

Example outputs you might see

- **If the element is visible (Fig 2)**

Fig 2

```
# The program continues execution, no output is printed unless explicitly coded
```

- **If the element is not visible in time** (after 10 seconds) **(Fig 3)**

Fig 3

```
Traceback (most recent call last):
  ...
selenium.common.exceptions.TimeoutException: Message
```

2 Use Timeouts to Handle Delays

- Set timeouts for tests that depend on network responses, external APIs, or services, ensuring tests don't fail prematurely.

- **Example in API Testing (Fig 4)**

Fig 4

```
import requests
response = requests.get('http://example.com/api', timeout=10)
```

Output (Fig 5)

Fig 5

```
# To check the status code
print(response.status_code)

# To see the response text (content)
print(response.text)

# To see headers in the response
print(response.headers)
```

3 Manage Test Dependencies

- Ensure that dependent tests are executed in the correct order, especially if one test must complete before another begins.
- **Example in pytest (Fig 6)**

Fig 6

```
@pytest.mark.dependency()
def test_first():
    assert some_setup()

@pytest.mark.dependency(depends=["test_first"])
def test_second():
    assert some_other_check()
```

Output

- **If both assertions pass (Fig 7)**

Fig 7

```
PASSED test_first
PASSED test_second
```

- **If test_first() fails (Fig 8)**

Fig 8

```
FAILED test_first
SKIPPED test_second
```

- **If test_second() fails but test_first() passes (Fig 9)**

Fig 9

```
PASSED test_first
FAILED test_second
```

4 Synchronize Multi-threaded or Parallel Tests

- Use synchronization mechanisms like locks or semaphores to manage access to shared resources in concurrent tests.
- **Example with threading Lock (Fig 10)**

Fig 10

```
import threading

# Create a Lock object
lock = threading.Lock()

# Function that ensures thread-safe execution
def thread_safe_function():
    with lock:  # Ensure only one thread can access this block at a time
        # Perform the test or some critical operation
        perform_test()

# Example function to simulate the test or task being executed
def perform_test():
    print("Test is being performed by thread:", threading.current_thread().name)

# Creating multiple threads to test thread safety
threads = []

for i in range(5):  # Create 5 threads for example
    thread = threading.Thread(target=thread_safe_function, name=f"Thread-{i}")
    threads.append(thread)
    thread.start()

# Wait for all threads to finish
for thread in threads:
    thread.join()
```

© NIMI NOT TO BE REPUBLISHED

Output (Fig 11)

Fig 11

```
Test is being performed by thread: Thread-0
Test is being performed by thread: Thread-1
Test is being performed by thread: Thread-2
Test is being performed by thread: Thread-3
Test is being performed by thread: Thread-4
```

5 Wait for External System Responses

- If your test interacts with an external system (e.g., database, API), ensure the system is ready and in the expected state before continuing the test.
- **Example of polling for a response (Fig 12)**

Fig 12

```python
import time

def wait_for_response():
    max_retries = 10  # Maximum number of retries
    for _ in range(max_retries):  # Loop through the retries
        if check_external_system():  # If the check is successful
            return True
        time.sleep(1)  # Wait for 1 second before retrying
    return False  # If all retries fail, return False

# Example function to simulate checking an external system
def check_external_system():
    # Simulating a check (this can be replaced with actual logic)
    # Let's say we randomly simulate success or failure:
    import random
    return random.choice([True, False])

# Running the function to check behavior
if wait_for_response():
    print("External system responded successfully.")
else:
    print("Failed to get a response after maximum retries.")
```

© NIMI NOT TO BE REPUBLISHED

Output

- **If check_external_system() succeeds within 10 retries (Fig 13)**

Fig 13

```
External system responded successfully.
```

- **If check_external_system() fails for all 10 retries (Fig 14)**

Fig 14

```
Failed to get a response after maximum retries.
```

6 Adjust for Timing Differences in UI Tests

- Ensure UI elements are fully loaded, visible, or interactable before performing actions like clicks or form submissions.
- **Example in Web Automation (Fig 15)**

Fig 15

```python
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# Wait for the element to become clickable
element = WebDriverWait(driver, 18).until(
    EC.element_to_be_clickable((By.ID, 'submit_button'))
)

# Click the element
element.click()
```

Output (Fig 16)

Fig 16

```
Traceback (most recent call last):
    ...
selenium.common.exceptions.TimeoutException: Message
```

Running the synchronized test

Objective: At the end of this exercise you shall be able to
- **execute tests with proper timing and dependencies.**

Requirements

Tool/Instruments
- A working PC - as reqd.

PROCEDURE

Running the synchronized test in **Synchronizing Tests** involves executing the tests while ensuring all timing and dependencies are properly handled. The goal is to ensure that the tests run in the correct order, with adequate waiting for necessary conditions, and without interference from race conditions or resource conflicts. Here's how to run synchronized tests effectively.

Key Steps for Running Synchronized Tests

1 Verify All Dependencies Are Resolved

- Before running tests, ensure that any dependent tests or operations (e.g., database state, external services) are completed and in the expected state.
- **Example in pytest (Fig 1)**

Fig 1

```
import pytest

@pytest.mark.dependency()
def test_setup():
    # Ensure setup is complete
    assert setup_is_done()  # Replace with actual setup logic

@pytest.mark.dependency(depends=["test_setup"])
def test_run():
    # Proceed only after setup is complete
    assert some_test_condition()  # Replace with actual test condition
```

© NIMI NOT TO BE REPUBLISHED

Scenario 1: Both test_setup() and test_run() pass

- **setup_is_done() returns True:** The setup is successful.
- **some_test_condition() returns True:** The test condition is met.

Output (Fig 2)

Fig 2

```
PASSED test_setup
PASSED test_run
```

Scenario 2: test_setup() fails (e.g., setup_is_done() returns False)

- **setup_is_done() returns False:** The setup fails.
- **test_run() is skipped** because it depends on test_setup().

Output (Fig 3)

Fig 3

```
FAILED test_setup
SKIPPED test_run
```

Scenario 3: test_run() fails (e.g., some_test_condition() returns False)

- **setup_is_done() returns True:** The setup is successful.
- **some_test_condition() returns False:** The test condition fails.

Output (Fig 4)

Fig 4

```
PASSED test_setup
FAILED test_run
```

2 Execute with Explicit Waits

- When running tests that depend on external systems (like UI elements or APIs), ensure explicit waits are in place to synchronize the test with the completion of these operations.
- **Example in Selenium (Fig 5)**

Fig 5

```
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
from selenium.common.exceptions import TimeoutException

try:
    # Wait for the element to become visible
    WebDriverWait(driver, 15).until(
        EC.visibility_of_element_located((By.ID, 'submit_button'))
    )

    # Click the submit button after it's visible
    driver.find_element(By.ID, 'submit_button').click()
    print("Button clicked successfully.")

except TimeoutException:
    print("The submit button did not become visible within the given time.")
```

Output (Fig 6)

Fig 6

```
Button clicked successfully.
```

3 Run Parallel Tests with Synchronization

- When running tests in parallel (e.g., in a continuous integration environment), ensure synchronization by controlling access to shared resources using locks, barriers, or semaphores.
- **Example with threading.Lock in Python (Fig 7)**

Fig 7

```
import threading
import time

lock = threading.Lock()

def perform_critical_section():
    print(f"Thread {threading.current_thread().name} is executing critical section")
    time.sleep(2)  # Simulating a time-consuming task
    print(f"Thread {threading.current_thread().name} has finished executing critical section")

def synchronized_test():
    with lock:
        perform_critical_section()

# Create multiple threads
threads = []
for i in range(5):
    t = threading.Thread(target=synchronized_test, name=f"Thread-{i+1}")
    threads.append(t)
    t.start()

# Wait for all threads to finish
for t in threads:
    t.join()
```

© NIMI NOT TO BE REPUBLISHED

Output (Fig 8)

Fig 8

```
Thread Thread-1 is executing critical section
Thread Thread-1 has finished executing critical section
Thread Thread-2 is executing critical section
Thread Thread-2 has finished executing critical section
Thread Thread-3 is executing critical section
Thread Thread-3 has finished executing critical section
Thread Thread-4 is executing critical section
Thread Thread-4 has finished executing critical section
Thread Thread-5 is executing critical section
Thread Thread-5 has finished executing critical section
```

4 Use Timeouts and Polling for External Operations

- When interacting with external systems (e.g., APIs, databases), use timeouts or polling to ensure operations are completed before proceeding with the next test step.
- **Example in API Testing (Fig 9)**

Fig 9

```python
import requests

# Sending a GET request with a timeout of 10 seconds
response = requests.get('http://example.com/api', timeout=10)

# Assert that the status code of the response is 200 (OK)
assert response.status_code == 200
```

Output (Fig 10)

Fig 10

```
Traceback (most recent call last):
  File "your_script.py", line 5, in <module>
    assert response.status_code == 200
AssertionError
```

5 Ensure UI Elements Are Ready

- For UI tests, ensure that elements are fully loaded, visible, and ready for interaction before proceeding with actions like clicking or form submission.
- **Example in Web Automation with WebDriverWait (Fig 11)**

Output (Fig 12)

Fig 12

```
Button clicked successfully.
```

6 Monitor for Asynchronous Test Completion

- For tests involving asynchronous operations, ensure that the test waits for the completion of background tasks or responses before asserting conditions.

Fig 11

```python
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
from selenium.common.exceptions import TimeoutException, NoSuchElementException

try:
    # Wait for the element to be clickable
    WebDriverWait(driver, 10).until(
        EC.element_to_be_clickable((By.ID, 'submit_button'))
    )

    # Click the button once it is clickable
    driver.find_element(By.ID, 'submit_button').click()
    print("Button clicked successfully.")

except TimeoutException:
    print("The submit button did not become clickable within the given time.")

except NoSuchElementException:
    print("The submit button was not found on the page.")
```

© NIMI NOT TO BE REPUBLISHED

- **Example with Polling (Fig 13)**

Fig 13

```python
import time

def wait_for_background_task():
    # Set the maximum number of retries
    max_retries = 5

    for _ in range(max_retries):
        # Check if the background task is complete
        if check_task_complete():
            return True

        # Wait for 1 second before checking again
        time.sleep(1)

    # If the task isn't complete after all retries, return False
    return False
```

Output

1 **If the background task completes in time (within 5 attempts),** the output will be, (Fig 14)

Fig 14

```
Background task completed successfully!
```

2 **If the background task does not complete within the allowed time (after 5 attempts),** the output will be, (Fig 15)

Fig 15

```
Background task did not complete in time.
```

7 Track Test Progress and Dependencies

- When running synchronized tests, particularly in parallel or complex workflows, monitor progress using logging or a test framework that tracks the completion of each step or dependency.

- **Example in Logging (Fig 16)**

Fig 16

```python
import logging

# Set up logging configuration with INFO level
logging.basicConfig(level=logging.INFO)

def run_synchronized_test():
    logging.info("Starting test execution...")

    # Execute test steps with synchronization logic
    # Example placeholder for test steps
    # For example, sleep or wait for elements, etc.

    logging.info("Test completed successfully.")
```

Example Output (Fig 17)

Fig 17

```
INFO:root:Starting test execution...
INFO:root:Test completed successfully.
```

© NIMI NOT TO BE REPUBLISHED

How do you check a bitmap?

Objective: At the end of this exercise you shall be able to
- **verify bitmap accuracy by comparing expected values.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

To **check a bitmap** in **Checking Bitmaps,** you compare the actual bitmap (an image file or screen snapshot) with an expected one to validate its accuracy. This is often done in image testing, UI validation, or automated visual testing scenarios, ensuring that the bitmap displays the correct visual content as intended.

Steps to Check a Bitmap

1 Capture the Bitmap

- Obtain the bitmap by either taking a screenshot of the application or by extracting the image from the relevant source (e.g., a UI element, web page, or graphic asset).

2 Compare with Expected Bitmap

- The captured bitmap is compared against a reference or expected bitmap that represents the correct or baseline image. This comparison can be done using pixel-by-pixel matching or more advanced image comparison algorithms.

Example

- Use tools like ImageMagick, Sikuli, or Applitools to compare the captured bitmap to the expected image.

3 Use Image Comparison Tools

- Pixel-by-Pixel Comparison: Compare each pixel of the actual bitmap to the expected bitmap. Any difference, even a single pixel, will be flagged as a mismatch.
- Fuzzy Matching: If the images are almost identical with minor visual differences (like anti-aliasing, slight UI changes, or dynamic content), fuzzy matching can be used to allow small deviations.

Fig 1

```
from PIL import Image
import imagehash

# Open images
actual_image = Image.open('actual_image.png')
expected_image = Image.open('expected_image.png')

# Calculate the hash for both images
actual_hash = imagehash.average_hash(actual_image)
expected_hash = imagehash.average_hash(expected_image)

# Compare hashes
if actual_hash == expected_hash:
    print("Bitmaps match!")
else:
    print("Bitmaps do not match!")
```

© NIMI NOT TO BE REPUBLISHED

Example using Python and ImageHash (Fig 1)

Example of Output

Fig 2

```
Bitmaps match!
```

- **If the images match (Fig 2)**

Fig 3

```
Bitmaps do not match!
```

- **If the images do not match (Fig 3)**

4 Handle Differences in Visual Elements

- **Tolerate Minor Differences:** If the bitmap check allows for minor differences (e.g., dynamic content, UI animations), configure the comparison tool to accept a predefined threshold of difference.

5 Check for Changes in Bitmap Attributes

- Verify attributes like image resolution, size, aspect ratio, and color consistency. This ensures that the bitmap not only matches visually but also meets technical specifications.

6 Automate Bitmap Checks in Test Suites

- Integrate bitmap checking in your test automation pipeline to verify UI changes, branding elements, and visual integrity in software applications.

Fig 4

```python
# Initialize WebDriver
driver = webdriver.Chrome()

try:
    # Open the Eyes visual testing session
    eyes.open(driver, "Test App", "Homepage")

    # Navigate to the desired URL
    driver.get("https://www.example.com")

    # Capture a screenshot and check the page (this is your "visual checkpoint")
    eyes.check_window("Homepage Bitmap")

finally:
    # Close the Eyes session and end the visual test
    eyes.close()

    # Quit the WebDriver session
    driver.quit()
```

© NIMI NOT TO BE REPUBLISHED

Example with Selenium and Applitools (Fig 4)

Fig 5

```
Traceback (most recent call last):
  File "visual_test.py", line 5, in <module>
    driver = webdriver.Chrome()
TypeError: 'NoneType' object is not callable
```

Adding bitmap checkpoints to a test script

Objective: At the end of this exercise you shall be able to
- **incorporate bitmap comparisons for visual validation checkpoints.**

Requirements

Tool/Instruments
- A working PC - as reqd.

PROCEDURE

Adding bitmap checkpoints to a test script in **Checking Bitmaps** involves integrating image comparison functionality at specific points in the script to validate the visual correctness of the application. These checkpoints ensure that the actual bitmap matches the expected visual output at various stages of the test, helping identify UI discrepancies.

Steps to Add Bitmap Checkpoints to a Test Script:

1 Capture the Current Bitmap

- During the test, capture a bitmap (screenshot or image) of the UI at the desired checkpoint.
- This bitmap represents the actual UI state at that moment.

Example in Selenium (Python) (Fig 1)

Fig 1

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('https://example.com')

# Capture screenshot
driver.save_screenshot('actual_screenshot.png')
```

Output (Fig 2)

Fig 2

```
Traceback (most recent call last):
  File "screenshot_script.py", line 2, in <module>
    driver = webdriver.Chrome()
TypeError: 'NoneType' object is not callable
```

2 Set Expected Bitmaps

- Store the expected bitmap (reference image) for comparison. This can be an image that you have previously captured or one provided by your design team.
- The expected bitmap should represent the correct visual state of the UI at that point.

Example Expected Bitmap

- expected_screenshot.png stored in your test assets directory.

3 Integrate Bitmap Comparison Logic

- Use an image comparison tool to compare the captured bitmap with the expected one.
- This can be done pixel-by-pixel or using more advanced algorithms that allow for small visual differences.

Example with Python and ImageHash (Fig 3)

Fig 3

```
from PIL import Image
import imagehash

# Load the actual and expected images
actual_image = Image.open('actual_screenshot.png')
expected_image = Image.open('expected_screenshot.png')

# Compare images using average hash
actual_hash = imagehash.average_hash(actual_image)
expected_hash = imagehash.average_hash(expected_image)

if actual_hash == expected_hash:
    print("Bitmap match successful!")
else:
    print("Bitmap mismatch detected!")
```

Expected Output

1 If the images match

- If actual_screenshot.png and expected_screenshot.png are visually identical, their average hashes will also be the same. The output will be, (Fig 4)

Fig 4

```
Bitmap match successful!
```

2 If the images differ

- If there are any differences between the two images (e.g., slight changes in layout, color, etc.), the hashes will differ, and the output will be, (Fig 5)

Fig 5

```
Bitmap mismatch detected!
```

© NIMI NOT TO BE REPUBLISHED

4 Add Checkpoints to the Test Script

- Insert these bitmap comparison steps at key points in the test to ensure that the UI displays the correct visual content at specific moments.
- These checkpoints can be added after critical actions (e.g., clicking a button, loading a page, or submitting a form).

Example Test Script with Checkpoints (Fig 6)

Fig 6

```
from selenium import webdriver
from PIL import Image
import imagehash

# Initialize the driver and navigate
driver = webdriver.Chrome()  # Fixed initialization
driver.get("https://example.com")

# Perform some actions
driver.find_element_by_id("submit_button").click()

# Capture screenshot after the action
driver.save_screenshot('actual_screenshot.png')

# Load the expected image
expected_image = Image.open('expected_screenshot.png')  # Fixed syntax

# Compare bitmaps
actual_image = Image.open('actual_screenshot.png')  # Fixed syntax
actual_hash = imagehash.average_hash(actual_image)
expected_hash = imagehash.average_hash(expected_image)

# Compare the hashes
if actual_hash == expected_hash:  # Fixed comparison
    print("Bitmap checkpoint passed!")
else:
    print("Bitmap checkpoint failed!")

# Quit the WebDriver session
driver.quit()
```

© NIMI NOT TO BE REPUBLISHED

Expected Output

1 If the bitmaps are identical (i.e., the screenshots match exactly) (Fig 7)

Fig 7

```
Bitmap checkpoint passed!
```

2 If the bitmaps differ (i.e., the screenshots have even the slightest difference) (Fig 8)

Fig 8

```
Bitmap checkpoint failed!
```

5 Handle Bitmap Comparison Failures

- If the bitmap comparison fails, log the difference or capture the failure in the test report.
- You can also highlight the areas where the differences occurred, or save the failed bitmap for further inspection.

6 Automate Bitmap Checking in CI/CD Pipeline

- Integrate these image comparison checks into your CI/CD pipeline to automatically validate visual consistency after each deployment or change.
- This ensures that any UI discrepancies are detected immediately.

Viewing expected results

Objective: At the end of this exercise you shall be able to
- **compare actual results with predefined expected bitmaps.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Viewing expected results in Checking Bitmaps involves comparing the actual captured bitmap (e.g., a screenshot or screen snapshot) to the predefined, reference image (expected result) to ensure visual consistency and correctness. This is crucial for validating UI elements and detecting any visual discrepancies, especially during automated testing.

Key Steps for Viewing Expected Results in Bitmap Checking

1 Obtain the Expected Bitmap

- The expected result is usually an image that represents the correct visual state of the UI, taken either manually or automatically in previous test runs.
- This image serves as a baseline reference to compare the actual results against.

Example

- A screenshot taken during an earlier test or design specification (e.g., expected_screenshot.png).

2 Capture the Actual Bitmap

- During the test execution, capture a screenshot of the current UI state after performing certain actions (e.g., clicking a button, loading a page, etc.).
- The captured image represents the "actual" result of the test.

Example in Selenium (Python) (Fig 1)

Fig 1

```python
from selenium import webdriver
from selenium.webdriver.common.by import By

# Step 1: Initialize the WebDriver (Chrome)
driver = webdriver.Chrome()

try:
    # Step 2: Navigate to the webpage
    driver.get("https://example.com")

    # Step 3: Perform actions (click a button with a specific ID)
    # Assuming there's a button with the ID "submit_button"
    driver.find_element(By.ID, "submit_button").click()

    # Step 4: Capture a screenshot and save it
    driver.save_screenshot('actual_screenshot.png')
    print("Screenshot saved as 'actual_screenshot.png'")

finally:
    # Step 5: Close the WebDriver (quit the browser)
    driver.quit()
```

© NIMI NOT TO BE REPUBLISHED

Output (Fig 2)

Fig 2

```
Screenshot saved as 'actual_screenshot.png'
```

3 Compare Expected and Actual Bitmaps

- Use image comparison techniques to match the expected bitmap with the actual captured bitmap.
- This can be done using tools or libraries that perform pixel-by-pixel comparison or using more advanced algorithms (e.g., fuzzy matching) that allow for minor visual differences.

Example using Python and imagehash (Fig 3)

Fig 3

```python
from PIL import Image
import imagehash

# Load the actual and expected images
actual_image = Image.open('actual_screenshot.png')  # Corrected variable assignment and string
expected_image = Image.open('expected_screenshot.png')  # Corrected string quotes

# Compare the images using image hashing
actual_hash = imagehash.average_hash(actual_image)  # Corrected function call
expected_hash = imagehash.average_hash(expected_image)  # Corrected function call

# Compare the hashes
if actual_hash == expected_hash:
    print("Expected result matches the actual result.")
else:
    print("Mismatch detected between expected and actual results.")
```

Output

- If the images match (i.e., the visual content is the same and the hashes are identical), you will get the following output (Fig 4)
- If the images don't match (even with a small difference), you will see. (Fig 5)

Fig 4

```
Expected result matches the actual result.
```

Fig 5

```
Mismatch detected between expected and actual results.
```

4 Viewing and Analyzing Differences

- If a mismatch occurs, you can analyze the differences either by inspecting the images manually or using tools that highlight discrepancies.
- Some tools, like Applitools, automatically highlight the areas where the images differ.

© NIMI NOT TO BE REPUBLISHED

Example with Applitools for visual comparison (Fig 6)

Fig 6

```python
from applitools.selenium import Eyes
from selenium import webdriver

# Initialize Eyes instance
eyes = Eyes()

# Initialize WebDriver
driver = webdriver.Chrome()

try:
    # Open the Eyes visual testing session
    eyes.open(driver, "Test App", "Homepage")

    # Navigate to the desired URL
    driver.get("https://www.example.com")

    # Capture a screenshot and check the page (this is your "visual checkpoint")
    eyes.check_window("Homepage Checkpoint")

finally:
    # Close the Eyes session and end the visual test
    eyes.close()

    # Quit the WebDriver session
    driver.quit()
```

5 Handle Visual Changes or Differences

- In some cases, slight differences may be allowed (e.g., dynamic content, animations, or minor UI updates). You can configure the comparison tool to tolerate small variations.
- Document or log significant differences for further investigation.

6 Automate Bitmap Checks in Test Suites

- Integrate the bitmap comparison steps into your automated test scripts to automatically validate UI consistency as part of your continuous integration (CI) pipeline.

© NIMI NOT TO BE REPUBLISHED

Running the test on a new version

Objective: At the end of this exercise you shall be able to
- **test new version by comparing updated bitmaps.**

Requirements	
Tool/Instruments	
• A working PC	- as reqd.

PROCEDURE

Running the test on a new version in **Checking Bitmaps** involves validating the visual integrity of the updated application by comparing the captured bitmaps (screenshots) of the new version with the expected results. This helps detect any unintended visual changes or regressions that may have occurred after code updates, UI redesigns, or version changes.

Steps to Run the Test on a New Version

1 Capture the Actual Bitmap of the New Version

- After deploying the new version of the application, run the test and capture the current UI state using screenshots.
- This captured bitmap will represent the actual UI state after the changes in the new version.

Example in Selenium (Python) (Fig 1)

Fig 1

```
import os
print(os.path.exists('new_version_actual.png'))
```

- **If the file exists,** it will output. (Fig 2)

Fig 2

```
True
```

- **If the file does not exist,** it will output. (Fig 3)

Fig 3

```
False
```

2 Compare with Expected Bitmap

- The captured bitmap from the new version needs to be compared with the expected bitmap (previous version or baseline image) to check for visual discrepancies.
- If you're running the test for the first time, the "expected bitmap" may be a version captured from the prior stable release.

Example using ImageHash for comparison (Fig 4)

Fig 4

```
from PIL import Image  # Correct import statement for the Python Imaging Library (PIL)
import imagehash

# Load the new version screenshot and the expected screenshot
actual_image = Image.open("new_version_actual.png")  # Correct the assignment with '='
expected_image = Image.open("expected_screenshot.png")  # Fixed quote issue

# Compare the images using image hashing
actual_hash = imagehash.average_hash(actual_image)  # Corrected assignment
expected_hash = imagehash.average_hash(expected_image)  # Corrected assignment

# Compare the image hashes and print results
if actual_hash == expected_hash:
    print("New version visual result matches expected.")
else:
    print("Mismatch detected between new version and expected bitmap.")
```

© NIMI NOT TO BE REPUBLISHED

Output

- If the hash of both images (actual_hash and expected_hash) is the same, the output will be (Fig 5)

Fig 5

```
New version visual result matches expected.
```

- If the hashes differ (meaning the images do not match), the output will be (Fig 6)

Fig 6

```
Mismatch detected between new version and expected bitmap.
```

3 Identify and Handle Differences

- If differences are detected, review and handle the changes.
  - **Small differences:** These may be acceptable, like changes in dynamic content, minor UI animations, or theme adjustments.
  - **Significant differences:** Larger discrepancies may indicate bugs or unintended changes that need to be fixed.

Use tools like Applitools for automatic visual testing (Fig 7)

Fig 7

```
from applitools.selenium import Eyes  # Correct import statement for Applitools Eyes
from selenium import webdriver

eyes = Eyes()  # Corrected the instantiation of Eyes object
driver = webdriver.Chrome()  # Corrected the instantiation of the Chrome WebDriver

try:
    eyes.open(driver, "Test App", "New Version Test")  # Open the Eyes session
    driver.get("https://www.example.com")  # Navigate to the webpage
    eyes.check_window("New Version Checkpoint")  # Capture a screenshot and compare to baseline
finally:
    eyes.close()  # Close the Eyes session
    driver.quit()  # Quit the WebDriver session
```

© NIMI NOT TO BE REPUBLISHED

Output

- **When running the code:** Applitools Eyes will capture a screenshot of the webpage and compare it to the baseline images you've previously uploaded to your Applitools account.
- If there are any visual differences between the baseline and the current screenshot, the result will be flagged as a visual issue.
- The result will not be printed to the console directly. Instead, you'll see the test results on your Applitools dashboard under the project "Test App" with the test name "New Version Test."

4 Update Expected Bitmaps if Necessary

- If the new version is intended to have visual changes, update the expected bitmaps to reflect the updated UI.
- Store the updated screenshots as the new baseline for future tests.

5 Automate Bitmap Comparison in CI/CD Pipeline

- Incorporate bitmap checking into your automated testing pipeline to continuously validate visual changes during deployment or after updates.
- This helps in early detection of any UI regressions or issues introduced in new versions.

Bitmap checkpoint tips

Objective: At the end of this exercise you shall be able to
- **ensure reliable visual checks with consistent bitmaps.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

When working with **Bitmap checkpoints,** particularly in the context of computing (like databases or logging systems), the term typically refers to a method of saving the state of a system or data at a specific point in time to prevent data loss or enable rollback. Bitmap checkpoints are often used in database systems to mark points in a log, enabling efficient recovery or incremental changes.

Bitmap Checkpointing Process Step by Step

If you're referring to **bitmap indexing** in databases, where a bitmap is used to represent a set of data (often boolean values), here's a guide on how the checkpointing process might look. Let's break it down step by step:

Step 1: Understand Bitmap Representation

A bitmap is an array of bits (0s and 1s) that can efficiently represent the presence or absence of items within a set.

- For example, a set {1, 4, 6, 7} could be represented with a bitmap as (Fig 1)

Fig 1

```
Bitmap: 1011001
```

Step 2: Create Initial Bitmap (Base) Representation

When you first create a bitmap, you need to determine what elements are part of your data set. If you want to track which elements are active, your bitmap could represent these as follows

1. Initialize the bitmap with all zeros. The size of the bitmap corresponds to the total number of elements in the data set.
2. Set the corresponding bits to 1 for any items in the set.

Example

If your data set is {1, 2, 3, 4, 5} and the items you want to mark are {1, 4}, the initial bitmap might look like this: (Fig 2)

Fig 2

```
Bitmap: 10010
```

Where

- 1 represents the presence of the item in the set.
- 0 represents its absence.

Step 3: Perform Operations & Modify Bitmap

You might add or remove elements from the set, which will require updating the bitmap. These operations are crucial for maintaining an accurate state in your bitmap.

Example

If you add 2 and remove 4, you would update the bitmap (Fig 3)

Fig 3

```
Before: 10010 (representing {1, 4})
After: 11100 (representing {1, 2, 5})
```

Step 4: Set a Checkpoint (Mark State)

A checkpoint is essentially the process of saving the current state of the bitmap so that you can refer back to it or restore it later.

- You save the bitmap as it is at the moment. This is useful in recovery systems or incremental backups.
- A bitmap checkpoint is simply a snapshot of the current bitmap at a particular point in time.

Example

Let's say we take a checkpoint after performing several updates to our dataset (Fig 4)

Fig 4

```
Checkpoint Bitmap: 11100
```

© NIMI NOT TO BE REPUBLISHED

This checkpoint ensures that if there's any failure, you can restore the bitmap to the state it was at the time of the checkpoint.

Step 5: Use Bitmap for Recovery

When restoring, you refer back to the checkpointed bitmap and reload it. You then can recover the dataset or system to the state it was in at that specific point in time.

Example

If the system crashes after an update, you could recover by restoring the checkpointed bitmap (Fig 5)

Fig 5

```
Restored Bitmap: 11100
```

Output

1 Initial Bitmap (Fig 6)

Fig 6

```
10010
```

2 After Operations

- **Add 2, Remove 4 (Fig 7)**

Fig 7

```
10010
```

3 Checkpoint Bitmap

- **Saved state (Fig 8)**

Fig 8

```
11100
```

4 Restore after Failure

- Restored from checkpoint (Fig 9)

Fig 9

```
11100
```

© NIMI
NOT TO BE REPUBLISHED

How do you program tests with TSL

Objective: At the end of this exercise you shall be able to
- **write and execute tests using TSL scripting.**

Requirements

Tool/Instruments
- A working PC - as reqd.

PROCEDURE

In **Programming Tests with TSL (Test Script Language),** you write and execute automated tests for applications by creating scripts using TSL syntax. TSL is a domain-specific language for writing test scripts, commonly used in test automation frameworks. The language allows testers to define, control, and validate application behavior in a structured way.

Steps to Program Tests with TSL

1 Set Up the Testing Environment
- Install the necessary testing framework that supports TSL. This may include tools like TestComplete (a popular test automation tool) that supports TSL scripting.
- Ensure the application under test (AUT) is accessible and properly configured for automation.

2 Create a New Test Script
- Begin by creating a new TSL test script file. In TestComplete, TSL is typically used for writing tests that interact with the application under test.

Example (Fig 1)

Fig 1

```
// A simple test script
function TestLogin()
{
    // Launch the application
    Sys.Process("MyApp").Run();

    // Find and interact with UI elements
    var usernameField = Aliases.MyApp.LoginForm.Username;
    usernameField.SetText("testuser");

    var passwordField = Aliases.MyApp.LoginForm.Password;
    passwordField.SetText("password123");

    var loginButton = Aliases.MyApp.LoginForm.LoginButton;
    loginButton.Click();

    // Verify the result
    var welcomeMessage = Aliases.MyApp.WelcomeForm.Message;
    if (welcomeMessage.Exists)
        Log.Message("Login successful!");
    else
        Log.Error("Login failed!");
}
```

Expected Output

The output will either be
- **"Login successful!"** if the WelcomeForm.Message exists (indicating that the login was successful).
- **"Login failed!"** if the message doesn't exist (indicating a failure in the login process).

3 Define Test Actions
- Use TSL's functions and commands to interact with the application's UI elements, perform actions (e.g., clicks, text entry), and validate results.
- TestComplete offers a rich set of built-in functions to interact with the system, like Click, SetText, VerifyExists, and Log.Message.

4 Use Variables and Control Flow
- You can define variables to store dynamic data and control flow using conditional statements (if, else), loops (for, while), and error handling mechanisms.

© NIMI NOT TO BE REPUBLISHED

Example (Fig 2)

Fig 2

```
// Conditional check in TSL
var isLoggedIn = false;

if (Aliases.MyApp.WelcomeForm.Exists)
{
    isLoggedIn = true;
}

if (isLoggedIn)
{
    Log.Message("Test Passed: User logged in successfully.");
}
else
{
    Log.Error("Test Failed: User login failed.");
}
```

Output

Based on the presence of the WelcomeForm, the output will be:

- If the WelcomeForm exists (indicating successful login) (Fig 3)

Fig 3

```
Test Passed: User logged in successfully.
```

- If the WelcomeForm does not exist (indicating login failure) (Fig 4)

Fig 4

```
Test Failed: User login failed.
```

5 Use Built-in Functions for Validation

- TSL supports built-in functions for validating application behavior. You can check properties of UI elements (like visibility, enabled state, and text content) to validate expected results.

Example (Fig 5)

Fig 5

```
var loginButton = Aliases.MyApp.LoginForm.LoginButton;
if (loginButton.Enabled)
{
    Log.Message("Login button is enabled.");
}
else
{
    Log.Error("Login button is not enabled.");
}
```

Output

- **If the Login Button is enabled (Fig 6)**

Fig 6

```
Login button is enabled.
```

- **If the Login Button is not enabled (Fig 7)**

Fig 7

```
Login button is not enabled.
```

6 Run the Test

- After writing the test script, run it using the test runner integrated with the tool (e.g., TestComplete). You can execute individual tests or entire test suites.

7 Handle Test Results

- After the test executes, check the results in the test report. If the test fails, you can use the logs to debug the issue.
- Use assertions to handle validation, like verifying that specific UI elements exist or match expected values.

8 Automate and Integrate with CI/CD

- You can integrate TSL-based tests into your Continuous Integration/Continuous Deployment (CI/CD) pipeline to run automated tests on every code change or deployment.

© NIMI NOT TO BE REPUBLISHED

Recording a basic test script

Objective: At the end of this exercise you shall be able to
- **record and generate basic test scripts automatically.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Recording a basic test script in Programming Tests with TSL involves using an automated testing tool like **TestComplete** to generate test scripts by recording interactions with the application. These recorded interactions are then converted into TSL (Test Script Language) code, which you can later modify or enhance for more complex testing scenarios.

Here's a step-by-step guide to **recording a basic test script in TSL**

Steps for Recording a Basic Test Script in TSL

1 Open the Test Automation Tool (e.g., TestComplete)

- Launch the tool that supports TSL (such as TestComplete).
- Ensure that the application you want to test is running or accessible.

2 Start the Test Recording

- Navigate to the Record option in the tool's interface.
- The tool will begin recording all actions you perform on the application.

Example

- In TestComplete: Click on the "Record" button to start capturing actions like clicking buttons, typing text, and navigating through UI elements.

3 Perform Actions on the Application

- Interact with the application as a typical user would.
- Examples of actions include.
  - Entering text into input fields.
  - Clicking buttons.
  - Selecting items from dropdown menus.
  - Navigating through pages or forms.

4 Stop the Recording

- Once you have completed the necessary interactions, stop the recording.
- The tool will automatically generate the corresponding TSL script based on the actions you performed.

5 Review and Edit the Generated Script

- After recording, the tool will show you the automatically generated TSL code.
- You can inspect the code, which may look something like this (Fig 1)

Fig 1

```
function TestLogin()
{
    // Launch the application
    Sys.Process("MyApp").Run();

    // Enter username
    Aliases.MyApp.LoginForm.Username.SetText("testuser");

    // Enter password
    Aliases.MyApp.LoginForm.Password.SetText("password");

    // Click login button
    Aliases.MyApp.LoginForm.LoginButton.Click();

    // Verify login success
    if (Aliases.MyApp.HomePage.Exists)
    {
        Log.Message("Login successful!");
    }
    else
    {
        Log.Error("Login failed.");
    }
}
```

© NIMI NOT TO BE REPUBLISHED

Example (TSL Code Generated from Recording)

Output

1 **If the HomePage exists (login successful) (Fig 2)**

Fig 2

```
Login successful!
```

2 **If the HomePage does not exist (login failed) (Fig 3)**

Fig 3

```
Login failed.
```

6 **Modify the Script (if necessary)**

- If the recorded script requires adjustments or additional steps (like adding validation or handling dynamic content), you can manually edit the TSL code.

7 **Run the Recorded Test Script**

- Once you are satisfied with the recorded script, run it to see how it behaves.
- The tool will execute the recorded actions on the application and check for expected results.

8 **Integrate the Test into the Test Suite**

- If the test passes successfully, you can save it as part of your test suite.
- Optionally, you can set up automated testing schedules or integrate the script into your Continuous Integration (CI) pipeline.

© NIMI
NOT TO BE REPUBLISHED

Using the function generator to insert functions

Objective: At the end of this exercise you shall be able to
- **use function generator to insert test functions.**

Requirements	
Tool/Instruments	
• A working PC	- as reqd.

PROCEDURE

Using the Function Generator to Insert Functions in Programming Tests with TSL allows you to quickly generate reusable functions that can be integrated into your test scripts. The f**unction generator** tool typically offered by test automation frameworks like **TestComplete** helps you create custom functions to perform repetitive or complex tasks within your test scripts.

Here's how to use the Function Generator in T**estComplete (with TSL)**

Steps for Using the Function Generator to Insert Functions

1 Open TestComplete

- Ensure your test environment is set up, and your application under test (AUT) is running.
- Open TestComplete where your test project is located.

2 Access the Function Generator

- In TestComplete, the Function Generator is available in the toolbar or can be accessed through the Script Editor.
- Look for options like "Insert Function" or "Function Generator" in the toolbar or through the right-click context menu in the Script Editor.

3 Select Function Type

- The Function Generator allows you to create different types of functions, such as
  - **Standard Functions:** To encapsulate reusable actions or checks.
  - **Custom Functions:** For specialized logic or more complex operations.
  - Choose the function type you need, based on the actions you want to automate in your test.

4 Define Function Name and Parameters

- Enter a meaningful name for the function, such as TestLogin or VerifyElementVisibility.
- Optionally, define any parameters for the function, like input values for the function or specific data objects.

5 Write the Function Logic

- The Function Generator may allow you to define simple logic or select actions directly within the generator interface (like performing clicks or setting text).
- For example, a function to log in to the application may include actions to fill in the username, password, and click the login button.

© NIMI NOT TO BE REPUBLISHED

Example (Fig 1)

Fig 1

```
function TestLogin(username, password) {
    // Enter credentials
    Aliases.MyApp.LoginForm.Username.SetText(username);
    Aliases.MyApp.LoginForm.Password.SetText(password);

    // Click the login button
    Aliases.MyApp.LoginForm.LoginButton.Click();

    // Add a delay or wait for the next screen or message
    Delay(2);  // Assuming the transition takes a couple of seconds

    // Check for successful login
    if (Aliases.MyApp.MainForm.Exists) {
        Log.Message("Login successful!");
        return true;
    } else {
        Log.Message("Login failed!");
        return false;
    }
}
```

Output

Case 1: Successful Login (Fig 2)

Fig 2

```
Login successful!
```

Case 2: Failed Login (Fig 3)

Fig 3

```
Login failed!
```

6 Insert the Function

- After defining the function, click Insert to automatically add it to your test script.
- The generated function will appear at the appropriate location in the TSL script.

7 Use the Function in Your Test Script

- Once inserted, you can now call this function anywhere within your test script.
- Example usage (Fig 4)

Fig 4

```
// Call the TestLogin function with parameters
TestLogin("testuser", "password123");
```

Output (Fig 5)

Fig 5

```
Login successful!
```

8 Modify and Enhance the Function

- After inserting, you may want to further customize the function by adding assertions, logging, or handling specific test conditions.
- You can always edit the function later in the Script Editor to suit your testing needs.

9 Run the Test

- Execute the test script to see the functions in action. The function you created will be invoked as part of the script's execution.

© NIMI NOT TO BE REPUBLISHED

Example of a Function Generated Using the Function Generator (Fig 6)

Fig 6

```
// Function for logging in
function TestLogin(username, password) {
    Aliases.MyApp.LoginForm.Username.SetText(username);
    // Set username
    Aliases.MyApp.LoginForm.Password.SetText(password);
    // Set password
    Aliases.MyApp.LoginForm.LoginButton.Click();
    // Click login

    // Wait for a while to ensure the next screen or message is loaded
    Delay(2);  // 2-second delay to allow the app to transition

    // Check for successful login (e.g., by checking if the main form exists)
    if (Aliases.MyApp.MainForm.Exists) {
        Log.Message("Login successful!");  // Log success
    } else {
        Log.Message("Login failed!");  // Log failure
    }
}

// Use the TestLogin function in a test
TestLogin("testuser", "password123");
```

Output (Fig 7)

Fig 7

```
Login successful!
```

© NIMI NOT TO BE REPUBLISHED

Adding logic to the test scrip

Objective: At the end of this exercise you shall be able to
- **enhance test scripts with conditional and logical operations.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Adding logic to a test script in TSL (Test Script Language) allows you to introduce dynamic behavior, conditions, and decision-making processes within your automated tests. By incorporating logic such as **if/else** conditions, loops, and error handling, you can create more flexible, robust, and adaptable test scripts.

Steps for Adding Logic to the Test Script in TSL

1 Conditionals (If/Else Statements)

- Use **if/else** statements to execute specific actions based on certain conditions or results in your test.

Example (Fig 1)

Fig 1

```
function VerifyLogin()
{
    // Set username and password
    Aliases.MyApp.LoginForm.Username.SetText("testuser");
    Aliases.MyApp.LoginForm.Password.SetText("password123");
    Aliases.MyApp.LoginForm.LoginButton.Click();

    // Conditional check for login success
    if (Aliases.MyApp.HomePage.Exists)
    {
        Log.Message("Login successful.");
    }
    else
    {
        Log.Error("Login failed.");
    }
}
```

NOT TO BE REPUBLISHED

Expected Output

- If the login is successful (i.e., the home page is displayed after login), the output will be (Fig 2)

Fig 2

```
Login successful.
```

- If the login fails (i.e., the home page does not appear after login), the output will be (Fig 3)

Fig 3

```
Login failed.
```

2 **Loops (For/While Loops)**

- Use loops to repeat actions multiple times, such as verifying multiple UI elements or performing a task for a number of iterations.

Example (Fig 4)

Fig 4

```
function VerifyMultipleLinks() {
    var links = Aliases.MyApp.HomePage.Links; // Assuming this is a collection of links

    // Loop through each link and verify its visibility
    for (var i = 0; i < links.Count; i++) {
        if (links[i].Exists) {
            Log.Message("Link " + i + " is visible.");
        } else {
            Log.Error("Link " + i + " is not visible.");
        }
    }
}
```

Expected Output

The function will loop through each link in the Links collection and log whether the link is visible (i.e., whether Exists is true or false).

- **If the link is visible** (i.e., links[i].Exists is true), the output will be (Fig 5)

Fig 5

```
Link 0 is visible.
Link 1 is visible.
Link 2 is visible.
...
```

- **If the link is not visible** (i.e., links[i].Exists is false), the output will be (Fig 6)

Fig 6

```
Link 0 is not visible.
Link 1 is not visible.
Link 2 is not visible.
...
```

3 **Switch/Case Statements**

- In situations with multiple distinct cases, a switch statement allows you to handle different scenarios based on a single value.

Example (Fig 7)

Fig 8

```
function CheckStatus(status) {
    switch (status) {
        case "Success":
            Log.Message("Test Passed.");
            break;
        case "Failure":
            Log.Error("Test Failed.");
            break;
        default:
            Log.Warning("Unknown status.");
            break;  // Adding the missing break to properly exit the switch statement
    }
}
```

© NIMI NOT TO BE REPUBLISHED

Output

- If status is "Success", the function logs (Fig 8)

Fig 8

```
Test Passed.
```

- If status is "Failure", the function logs (Fig 8)

Fig 9

```
Test Failed.
```

- For any other value of status, the function logs (Fig 10)

Fig 10

```
Unknown status.
```

4 Error Handling (Try/Catch)

- Use try/catch blocks to handle exceptions and ensure that your tests continue running smoothly, even if an error occurs.

Example (Fig 11)

Fig 11

```
function SafeLogin(username, password) {
    try {
        Aliases.MyApp.LoginForm.Username.SetText(username);
        Aliases.MyApp.LoginForm.Password.SetText(password);
        Aliases.MyApp.LoginForm.LoginButton.Click();
    } catch (e) {
        // Handle error and log it
        Log.Error("Error during login: " + e.message);
    }
}
```

Output (Fig 12)

Fig 12

```
Error during login: Element not found.
```

5 Assertions

- Assertions are used to verify that the expected conditions are met. If the condition fails, the test will stop or log an error.

Example (Fig 13)

Fig 13

```
function VerifyTitle() {
    var title = Aliases.MyApp.MainWindow.Caption;  // Use '=' for assignment

    // Assert that the title is as expected
    if (title != "Expected Title") {
        Log.Error("Title mismatch: Expected 'Expected Title', got '" + title + "'");
    } else {
        Log.Message("Title verified.");
    }
}
```

© NIMI NOT TO BE REPUBLISHED

Output

- **If the title is not equal to "Expected Title"**
  - The log will show an error message like **(Fig 14)**

Fig 14

```
Title mismatch: Expected 'Expected Title', got 'Actual Title'.
```

- **If the title matches "Expected Title"**
  - The log will show

Fig 15

```
Title verified.
```

6 Variables and Data Management

- Use variables to store dynamic data and make decisions based on that data. Variables can store values like text, numbers, or object references.

Example (Fig 16)

Fig 16

```
function SelectUserBasedOnRole(role) {
    var username;

    if (role == "Admin") {
        username = "adminUser";
    } else if (role == "User") {
        username = "normalUser";
    } else {
        username = "guestUser";
    }

    // Set the username on the login form
    Aliases.MyApp.LoginForm.Username.SetText(username);

    // Log the username being used for login
    Log.Message("Logging in as: " + username);
}
```

© NIMI NOT TO BE REPUBLISHED

Output

If the role is "Admin", the log will show (Fig 17)

Fig 17

```
Logging in as: adminUser
```

If the role is "User", the log will show (Fig 18)

Fig 18

```
Logging in as: normalUser
```

If the role is neither "Admin" nor "User", the log will show (Fig 19)

Fig 19

```
Logging in as: guestUser
```

7 Functions and Reusability

- Create reusable functions to avoid code duplication. Logic in functions can be invoked multiple times throughout the script.

Example (Fig 20)

Fig 20

```
function Login(username, password) {
    Aliases.MyApp.LoginForm.Username.SetText(username);
    Aliases.MyApp.LoginForm.Password.SetText(password);
    Aliases.MyApp.LoginForm.LoginButton.Click();

    // Optional: Wait for the login process to complete (e.g., wait for the main form)
    Delay(2);  // Adjust the delay as needed based on your app's responsiveness

    // Check if the main form appears (assuming the MainForm indicates successful login)
    if (Aliases.MyApp.MainForm.Exists) {
        Log.Message("Login successful as: " + username);
    } else {
        Log.Error("Login failed for user: " + username);
    }
}

function TestLogin() {
    Log.Message("Starting TestLogin...");
    Login("testuser", "password123");  // Test login with standard user credentials
}

function TestAdminLogin() {
    Log.Message("Starting TestAdminLogin...");
    Login("admin", "adminpass");  // Test login with admin credentials
}
```

Output

When running TestLogin or TestAdminLogin, you will see something like

1 For TestLogin with standard user credentials ("testuser") (Fig 21)

Fig 21

```
Starting TestLogin...
Login successful as: testuser
```

2 For TestAdminLogin with admin credentials ("admin") (Fig 22)

Fig 22

```
Starting TestAdminLogin...
Login successful as: admin
```

© NIMI NOT TO BE REPUBLISHED

Understanding TL-step

Objective: At the end of this exercise you shall be able to
- **understand and utilize TL-step for test steps.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

What is a TL-step?

A **TL-step** (Test Log step) in **TestComplete** is simply a **log entry** for a specific action performed during your automated test. Each time you perform an action (like clicking a button, entering text, verifying an element's existence, etc.), a log entry is created. This log helps track the test's flow and outcome, which is very useful for debugging, reporting, and understanding test execution.

Basic Structure of a Test Log Step

In **TestComplete** (TSL), you generally log actions in a script with methods like Log.Message(), Log.Error(), Log.Warning(), and Log.AppendFolder().

Step-by-Step Example with Output

Here's a **step-by-step breakdown** of a TSL script with **Test Log steps (TL-steps)**

Test Case: Login Test

We'll simulate logging into an application using a username and password. (Fig 1)

© NIMI NOT TO BE REPUBLISHED

Fig 1

```
function LoginTest() {
  // Step 1: Start the Test Log
  Log.AppendFolder("LoginTest");  // This starts a new folder to organize the log entries

  // Step 2: Enter username
  Log.Message("Entering username 'testuser'");
  Aliases.MyApp.LoginForm.Username.SetText("testuser");

  // Step 3: Enter password
  Log.Message("Entering password 'password123'");
  Aliases.MyApp.LoginForm.Password.SetText("password123");

  // Step 4: Click the login button
  Log.Message("Clicking the login button");
  Aliases.MyApp.LoginForm.LoginButton.Click();

  // Step 5: Check if the login was successful
  if (Aliases.MyApp.MainForm.Exists) {
    Log.Message("Login successful.");
  } else {
    Log.Error("Login failed.");
  }

  // Step 6: End the Test Log
  Log.PopFolder();  // This ends the folder and finalizes the log entry for this test case
}
```

Breakdown of the Steps with Expected Output

Step 1: Start the Test Log (Fig 2)

Fig 2

```
Log.AppendFolder("LoginTest");
```

- **Purpose:** This creates a folder in the Test Log specifically for the "LoginTest" case. All subsequent log entries related to this test case will be organized under this folder.

Expected Output

- The test log will show (Fig 3)

Fig 3

```
[Test Log] - LoginTest
```

Step 2: Enter Username (Fig 4)

Fig 4

```
Log.Message("Entering username 'testuser'");
Aliases.MyApp.LoginForm.Username.SetText("testuser");
```

- **Purpose:** Logs a message that the script is entering the username "testuser". The script then sets the value of the username field to "testuser" on the login form.

Expected Output

- The log will show (Fig 5)

Fig 5

```
[Test Log] - Entering username 'testuser'
```

Step 3: Enter Password (Fig 6)

Fig 6

```
Log.Message("Entering password 'password123'");
Aliases.MyApp.LoginForm.Password.SetText("password123");
```

- **Purpose:** Logs the action of entering the password "password123". The script then sets the value of the password field to "password123".

Expected Output

- The log will show (Fig 7)

Fig 7

```
[Test Log] - Entering password 'password123'
```

Step 4: Click the Login Button (Fig 8)

Fig 8

```
Log.Message("Clicking the login button");
Aliases.MyApp.LoginForm.LoginButton.Click();
```

- **Purpose:** Logs the action of clicking the login button on the login form. It then performs the click action.

Expected Output

- The log will show (Fig 9)

Fig 9

```
[Test Log] - Clicking the login button
```

Step 5: Check if the Login Was Successful (Fig 10)

Fig 10

```
if (Aliases.MyApp.MainForm.Exists) {
  Log.Message("Login successful.");
} else {
  Log.Error("Login failed.");
}
```

- **Purpose:** The script checks if the main form (MainForm) exists, which would indicate a successful login. If it exists, it logs "Login successful." If it doesn't exist, it logs an error message "Login failed.".

Expected Output

- If the login is successful (Fig 11)

Fig 11

```
[Test Log] - Login successful.
```

- If the login fails (Fig 12)

Fig 12

```
[Test Log] - Login failed.
```

Step 6: End the Test Log (Fig 13)

Fig 13

```
Log.PopFolder();
```

- **Purpose:** This closes the folder "LoginTest" that was started earlier, marking the end of the test case and finalizing the log entries for this test.

Expected Output

- The log will indicate that the folder has been closed, finalizing the log for the test case. (Fig 14)

Fig 14

```
[Test Log] - End of LoginTest
```

© NIMI NOT TO BE REPUBLISHED

Debugging the test script

Objective: At the end of this exercise you shall be able to
- **identify and fix errors in test scripts.**

Requirements	
Tool/Instruments	
• A working PC	- as reqd.

PROCEDURE

Debugging the Test Script in Programming Tests with TSL is an essential skill to ensure that your automated tests run correctly and efficiently. Debugging helps identify, isolate, and fix issues in the test script that might cause failures, unexpected behavior, or inaccuracies in the test results. This process often involves using built-in debugging tools, logging, breakpoints, and analyzing the test execution flow.

Key Steps for Debugging a Test Script in TSL

1 Use Breakpoints

- Breakpoints allow you to pause the execution of your script at specific lines. This lets you inspect the values of variables, objects, and application states at that point in time.
- In TestComplete, you can set breakpoints in your TSL script by clicking the left margin of the line in the script editor where you want to pause.

Example (Fig 1)

Fig 1

```
function TestLogin()
{
  // Step 1: Start the application
  Log.Step("Start Application");
  Sys.Process("MyApp").Run();

  // Step 2: Set Username
  Log.Step("Set Username");
  Aliases.MyApp.LoginForm.Username.SetText("testuser");

  // Step 3: Set Password
  Log.Step("Set Password");
  Aliases.MyApp.LoginForm.Password.SetText("password123");

  // Optionally, you could set a breakpoint here for debugging if required.
}
```

© NIMI NOT TO BE REPUBLISHED

Output (Fig 2)

Fig 2

```
[Test Log]
   Start Application
```

- Once the breakpoint is hit, the script will pause, allowing you to check the values and behavior of the system.

2 Use Debugging Tools in TestComplete

- **Watch:** You can watch variables or expressions as the script executes, providing real-time information on how values change.
- **Call Stack:** The call stack shows the sequence of function calls leading up to the current point in the script. It helps identify where the test might be failing.
- **Step Through Execution:** Using "Step Into", "Step Over", and "Step Out" options, you can control the execution flow and observe the results of each action step by step.

3 Examine Test Logs

- The test log provides detailed information about each step of the test, including success or failure messages, errors, and additional diagnostic information.
- Review the test log to identify specific steps where the test failed or did not behave as expected.

- **Example (Fig 3)**

Fig 3

```
if (Aliases.MyApp.HomePage.Exists)
{
    Log.Message("Login successful.");
}
else
{
    Log.Error("Login failed.");
}
```

Output (Fig 4)

Fig 4

```
[Test Log]
   Login successful.
```

- If the test fails, you will get an error message in the log that shows where the issue occurred.

4 Use Print Statements or Logging

- Add log messages or print statements to your script to display the value of variables, results of conditions, or status of actions.
- These log entries help you trace the flow of the test and identify the point where things go wrong.

Example (Fig 5)

Fig 5

```
function TestLogin()
{
    Log.Message("Starting test...");
    tl-step("Open Application")
    Sys.Process("MyApp").Run();

    Log.Message("Setting username and password.");
    tl-step("Enter Username")
    Aliases.MyApp.LoginForm.Username.SetText("testuser");
    tl-step("Enter Password")
    Aliases.MyApp.LoginForm.Password.SetText("password123");

    if (Aliases.MyApp.HomePage.Exists)
    {
        Log.Message("Login successful.");
    }
    else
    {
        Log.Error("Login failed.");
    }
}
```

© NIMI NOT TO BE REPUBLISHED

Output (Fig 6)

Fig 6

```
[Test Log]
   Starting test...
   Open Application
   Setting username and password.
   Enter Username
   Enter Password
   Login successful.
```

5 Check for Common Errors

- **Object recognition issues:** If the script fails to find a UI element, ensure the object name and property are correct.
- **Timing issues:** Sometimes, scripts fail because actions are attempted before the application is ready. Use Wait functions or increase timeouts for better synchronization.

Example (Fig 7)

Fig 7

```
// Wait until the element is visible
Aliases.MyApp.LoginForm.LoginButton.WaitProperty("Visible", true, 5000);
```

Output

Case 1: Button Becomes Visible

If the LoginButton becomes visible within 5 seconds, the script will continue to execute the next line of code. There will be no visible output for this wait unless you explicitly log it, but the test will proceed smoothly.

Example Output (no visible log unless explicitly added) (Fig 8)

Fig 8

```
[Test Log]
   (no explicit log)
```

Case 2: Button Does Not Become Visible in 5 Seconds

If the LoginButton does not become visible within the 5-second timeout, TestComplete will raise an error indicating that the property "Visible" was not true within the specified time.

Example Output (Fig 9)

Fig 9

```
[Test Log]
   [Error] Property 'Visible' of 'LoginButton' is not 'True' after waiting for 5000 ms
```

6 Modify the Script for Error Handling

- Add try/catch blocks to handle exceptions that might occur during execution and provide more informative error messages.

© NIMI NOT TO BE REPUBLISHED

Example (Fig 10)

Fig 10

```
function TestLogin()
{
    try
    {
        tl-step("Open Application");
        Sys.Process("MyApp").Run();

        tl-step("Enter Username");
        Aliases.MyApp.LoginForm.Username.SetText("testuser");

        tl-step("Enter Password");
        Aliases.MyApp.LoginForm.Password.SetText("password123");

        if (Aliases.MyApp.HomePage.Exists)
        {
            Log.Message("Login successful.");
        }
        else
        {
            Log.Error("Login failed.");
        }
    }
    catch (e)
    {
        Log.Error("Error encountered: " + e.message);  // Fixed the concatenation
    }
}
```

Output

Case 1: Successful Login

If the login is successful (i.e., HomePage.Exists is true), the log will show (Fig 11)

Fig 11

```
[Test Log]
   Open Application
   Enter Username
   Enter Password
   Login successful.
```

Case 2: Failed Login

If the login fails (i.e., HomePage.Exists is false), the log will show (Fig 12)

Case 3: Error During Execution

If an error occurs during any step (for example, if there's an issue with the application, or an element doesn't exist or is not accessible), the log will show:

Fig 12

```
[Test Log]
   Open Application
   Enter Username
   Enter Password
   [Error] Login failed.
```

Fig 13

```
[Test Log]
   Open Application
   Enter Username
   Enter Password
   [Error] Error encountered: <specific error message>
```

7 Test with Different Data

- Sometimes, test failures occur due to unexpected data. Modify input data or test cases to check different scenarios, edge cases, or invalid inputs.

Example (Fig 14)

Fig 14

```
TestLogin("invaliduser", "wrongpassword");  // Testing with incorrect credentials
```

© NIMI NOT TO BE REPUBLISHED

Running the test on a new version

Objective: At the end of this exercise you shall be able to
- **test and validate scripts on new versions.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Running the Test on a New Version in Programming Tests with TSL involves executing your existing test scripts on a new version of the application to ensure that the application still behaves as expected after updates or changes. This process is crucial for **regression testing** and verifying that new changes do not break existing functionality.

Steps for Running Tests on a New Version in TSL

1 Update the Application Version

- Before running the test on the new version, ensure that the application has been updated to the latest version (either through an installer, deployment, or version control).
- Ensure that you have access to the latest build or executable of the application under test.

2 Update Test Data or Configurations (if needed)

- If the new version introduces changes to the application's UI, functionality, or data, update the test data and configurations accordingly. This includes changing object names, locators, or other environment settings.
- **Example:** If the login page layout has changed in the new version, you may need to adjust object recognition properties in your script.

Example (Fig 1)

Fig 1

```
function LoginTest()
{
    // Adjusted for new version UI changes
    Aliases.MyApp.NewLoginForm.Username.SetText("testuser");
    Aliases.MyApp.NewLoginForm.Password.SetText("password123");
    Aliases.MyApp.NewLoginForm.LoginButton.Click();
}
```

Output (Fig 2)

Fig 2

```
Total Tests Run: 1
Tests Passed: 1
Tests Failed: 0
Execution Time: 1.25 seconds
```

3 Verify Object Mapping and Test Scripts

- Ensure that all objects and controls in your test scripts are still correctly mapped to the new version of the application.
- Use the Object Spy tool in TestComplete to verify that the object names and properties have not changed, or update them if needed.

4 Run the Test on the New Version

- Execute your existing test scripts against the new version of the application.
- Ensure that the script is correctly interacting with the updated application and that the expected results are returned.

© NIMI NOT TO BE REPUBLISHED

Example (Fig 3)

Fig 3

```
function TestLoginOnNewVersion()
{
    // Running the test with updated object names and properties
    tl-step("Open Application")
    Sys.Process("MyApp").Run();

    tl-step("Enter Username")
    Aliases.MyApp.NewLoginForm.Username.SetText("testuser");

    tl-step("Enter Password")
    Aliases.MyApp.NewLoginForm.Password.SetText("password123");

    tl-step("Click Login Button")
    Aliases.MyApp.NewLoginForm.LoginButton.Click();

    if (Aliases.MyApp.NewHomePage.Exists)
    {
        Log.Message("Login successful on new version.");
    }
    else
    {
        Log.Error("Login failed on new version.");
    }
}
```

Output

1 If the login is successful (Fig 4)

Fig 4

```
Step 1: Open Application
    Action: Sys.Process("MyApp").Run()
    Status: Passed

Step 2: Enter Username
    Action: SetText("testuser") on Username field
    Status: Passed

Step 3: Enter Password
    Action: SetText("password123") on Password field
    Status: Passed

Step 4: Click Login Button
    Action: Click() on Login button
    Status: Passed

Step 5: Verify Login Success
    Action: Check if NewHomePage.Exists
    Status: Passed

Login successful on new version.
Test Summary:
--------------
Total Steps: 5
Passed: 5
Failed: 0
Execution Time: 2.35 seconds
```

2 If the login fails (Fig 5)

Fig 5

```
Step 1: Open Application
    Action: Sys.Process("MyApp").Run()
    Status: Passed

Step 2: Enter Username
    Action: SetText("testuser") on Username field
    Status: Passed

Step 3: Enter Password
    Action: SetText("password123") on Password field
    Status: Passed

Step 4: Click Login Button
    Action: Click() on Login button
    Status: Passed

Step 5: Verify Login Success
    Action: Check if NewHomePage.Exists
    Status: Failed
    Error: NewHomePage not found.

Login failed on new version.
Test Summary:
--------------
Total Steps: 5
Passed: 4
Failed: 1
Execution Time: 2.55 seconds
```

© NIMI NOT TO BE REPUBLISHED

5 Review Test Results

- After running the tests, check the test results to verify that the new version works as expected.
- Examine the logs for any failures or unexpected behavior.
- Pay special attention to any broken tests or regression issues.

6 Address New Issues

- If the new version introduced any issues, update your test scripts to account for them. This could involve adapting to new UI elements, changes in business logic, or new workflows.
- **Example:** If a new field is added in the UI, you may need to modify the test to interact with the new field.

7 Perform Regression Testing

- It's important to run regression tests to ensure that the changes in the new version haven't introduced unintended side effects in other parts of the application.
- Re-run your suite of tests on the new version and verify that all areas of the application are still functioning as expected.

© NIMI
NOT TO BE REPUBLISHED

How do you create data- driven tests?

Objective: At the end of this exercise you shall be able to
- **use external data to drive test execution.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Creating Data-Driven Tests involves using external data sources (like Excel files, databases, or CSVs) to drive the execution of your test scripts. This allows you to run the same test with different input data, providing more comprehensive test coverage and improving efficiency.

Here's a step-by-step guide to creating **data-driven tests** in **TestComplete**

1 Choose Your Data Source

The first step in creating data-driven tests is to choose the external data source that will provide the test data. Common sources include

- Excel spreadsheets
- CSV files
- Databases
- XML files

The data source should contain the various input values that your tests will use during execution.

2 Prepare Your Test Data

- Create a table or file that includes all the test data you need. Each row typically represents a different set of input values, and each column represents a specific test parameter.

Example (CSV File) (Fig 1)

Fig 1

```
Username,Password
testuser1,password123
testuser2,password456
testuser3,password789
```

Output (example of processing or display) (Fig 2)

Fig 2

```
Username: testuser1, Password: password123
Username: testuser2, Password: password456
Username: testuser3, Password: password789
```

If you process the CSV file and display the data, it could look like this

3 Create the Test Script

You need to design your test script to execute multiple times with different data from the data source. The script will loop through the data source and use each row of data to run a specific test.

© NIMI NOT TO BE REPUBLISHED

Example (TSL Script) (Fig 3)

Fig 3

```
function DataDrivenLoginTest()
{
    // Define the path to the CSV data file
    var dataFile = "C:\\Tests\\TestData.csv";

    // Load the CSV data into a DataTable
    var dataTable = DataTable("CSV", dataFile);

    // Loop through each row of the data source
    while (dataTable.Next())
    {
        // Extract data from each column
        var username = dataTable.Value("Username");
        var password = dataTable.Value("Password");

        tl-step("Open Application");
        Sys.Process("MyApp").Run();

        tl-step("Enter Username");
        Aliases.MyApp.LoginForm.Username.SetText(username);

        tl-step("Enter Password");
        Aliases.MyApp.LoginForm.Password.SetText(password);

        tl-step("Click Login Button");
        Aliases.MyApp.LoginForm.LoginButton.Click();

        // Check if the HomePage exists, indicating a successful login
        if (Aliases.MyApp.HomePage.Exists)
        {
            Log.Message("Login successful with: " + username);
        }
        else
        {
            Log.Error("Login failed with: " + username);
        }
    }
}
```

© TUMI NOT TO BE REPUBLISHED

Output (Fig 4)

Fig 4

```
Login successful with: testuser1
Login failed with: testuser2
Login successful with: testuser3
```

Explanation

- **DataTable("CSV", dataFile)** loads the CSV file as a data table.
- **dataTable.Next()** loops through each row of the data.
- **dataTable.Value("Username")** and **dataTable.Value("Password")** extract the data for each test iteration.

4 Configure the Data Source

- In **TestComplete**, go to the **Data-Driven Testing** dialog or configure the test script manually to specify how the data source (e.g., CSV, Excel) is read during the test.
- You may need to specify the file location, sheet name (for Excel), and columns for data mapping.

5 Run the Data-Driven Test

Once your test script is set up, run it. TestComplete will execute the test for each data set from the data source, allowing you to validate multiple scenarios automatically.

6 Review the Results

After running the test, review the results in the Test Log. You will see separate results for each test iteration with different data inputs. This allows you to analyze how the application behaves with each set of data.

Converting your test to a data driven test

Objective: At the end of this exercise you shall be able to
- **modify test to use external data source.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Converting your test to a data-driven test involves modifying an existing test to use an external data source (such as a CSV file, Excel sheet, or database) for the test inputs. This allows you to run the same test with different sets of data, improving test coverage and efficiency.

Here are the steps to convert a standard test into a data-driven test

1 Identify the Test Parameters

- Review your existing test and identify which variables or inputs need to be replaced with data-driven values. For example, if you are testing a login functionality, the parameters might include the username and password.

Example (Original Test) (Fig 1)

Fig 1

```
function TestLogin()
{
    tl-step("Open Application")
    Sys.Process("MyApp").Run();

    tl-step("Enter Username")
    Aliases.MyApp.LoginForm.Username.SetText("testuser");

    tl-step("Enter Password")
    Aliases.MyApp.LoginForm.Password.SetText("password123");

    tl-step("Click Login Button")
    Aliases.MyApp.LoginForm.LoginButton.Click();

    if (Aliases.MyApp.HomePage.Exists)
    {
        Log.Message("Login successful.");
    }
    else
    {
        Log.Error("Login failed.");
    }
}
```

- In the above example, the username and password are hardcoded values.

© NIMI NOT TO BE REPUBLISHED

2 Choose Your Data Source

- Select an external data source (e.g., CSV, Excel, or database) that will supply the input values. This external data file will store different sets of values that you want to test.

Example (CSV File) (Fig 2)

Fig 2

```
Username, Password
testuser1, password123
testuser2, password456
testuser3, password789
```

3 Modify the Test to Use External Data

- Replace the hardcoded input values (like "testuser" and "password123") with references to the data source. You will need to modify your test script to read the data from the external file and loop through the values.

Converted Data-Driven Test (TSL Script) (Fig 3)

Fig 3

```
function DataDrivenLoginTest()
{
    // Define the data source file
    var dataFile = "C:\\Tests\\TestData.csv";
    var dataTable = DataTable("CSV", dataFile);

    // Loop through each row of the data source
    while (dataTable.Next())
    {
        // Extract data from each column
        var username = dataTable.Value("Username");
        var password = dataTable.Value("Password");

        tl-step("Open Application")
        Sys.Process("MyApp").Run();

        tl-step("Enter Username")
        Aliases.MyApp.LoginForm.Username.SetText(username);

        tl-step("Enter Password")
        Aliases.MyApp.LoginForm.Password.SetText(password);

        tl-step("Click Login Button")
        Aliases.MyApp.LoginForm.LoginButton.Click();

        if (Aliases.MyApp.HomePage.Exists)
        {
            Log.Message("Login successful with: " + username);
        }
        else
        {
            Log.Error("Login failed with: " + username);
        }
    }
}
```

Copy code
© TVM NOT TO BE REPUBLISHED

4 Define the Data Source in TestComplete

- DataTable is used to load external data into the script. You need to define the data source format and file path.
- In this example, we are using a CSV file as the data source. For Excel, you would change the format to "Excel" and provide the appropriate file path.

Note: You can also use the Data-Driven Testing dialog in TestComplete to configure the data source, making the setup more intuitive.

5 Run the Data-Driven Test

- After modifying the script, run it. The test will automatically fetch the data from the external source (CSV file) and execute the test using the values from each row.
- The test will repeat for each row in the CSV file, validating multiple sets of data in one execution.

6 Review Results

- After running the data-driven test, check the Test Log for the results. TestComplete will log the outcome of each iteration using the different sets of data.
- The log will indicate whether the login was successful or failed for each set of input data.

Example of the Test Log (Fig 4)

Fig 4

```
Login successful with: testuser1
Login successful with: testuser2
Login failed with: testuser3
```

© NIMI
NOT TO BE REPUBLISHED

Adding data to the data table

Objective: At the end of this exercise you shall be able to
- **add input data to the external data table.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Adding Data to the Data Table in data-driven tests involves populating an external data source (like a CSV, Excel, or database) with the input values that will be used during test execution. This enables the test to run with different sets of data, ensuring comprehensive coverage.

Here's how to add data to a data table for **data-driven tests** in TestComplete:

1 Create Your Data Source (e.g., CSV, Excel)

- Choose an external data source to store your test data. The most common options are CSV files or Excel spreadsheets. The data source should have columns representing different variables (e.g., username, password, product ID), and rows for different test cases.

Example (CSV Data Source) (Fig 1)

Fig 1

```
Username, Password
testuser1, password123
testuser2, password456
testuser3, password789
```

Example (Excel Data Source) (Fig 2)

Fig 2

Username	Password
testuser1	password123
testuser2	password456
testuser3	password789

2 Add Data to the Data Table (Using External Tools)

- To add data to your data table, you can either manually update the CSV or Excel file, or use a database management system to add rows into a database table.
- **For CSV/Excel:** Open the file in an editor like Excel or any text editor and add rows of data.
- **For Databases:** Use SQL queries to insert new rows into the database table where your test data is stored.

Example (Inserting Data into CSV): Add rows with different values (Fig 3)

Fig 3

```
Username, Password
testuser1, password123
testuser2, password456
testuser3, password789
testuser4, password000
testuser5, password999
```

3 Add Data Programmatically (Optional)

You can also populate a data table within your test script, particularly when working with custom test data or dynamically generated values. This is useful if you need to create data on the fly instead of loading it from an external file.

Example (Programmatically Adding Data to a Data Table) (Fig 4)

Fig 4

```
function AddDataToTable()
{
    var dataFile = "C:\\Tests\\TestData.csv";
    var dataTable = DataTable("CSV", dataFile);

    // Add a new row of data to the table
    dataTable.AddRow();
    dataTable.Value("Username", "newuser");
    dataTable.Value("Password", "newpassword123");

    // Save the updated data table
    dataTable.Save();
}
```

© NIMI NOT TO BE REPUBLISHED

4 Verify Data Integrity

- After adding data to the external data source, ensure the format is correct and the data is accessible by your test scripts. If you're using a CSV file, make sure there are no extra commas or spaces in the rows and columns.

Example (Verify CSV File): Open the CSV file in a text editor or Excel to ensure the values are correctly separated and aligned.

5 Update the Test Script

Once the data is added to the data source, make sure your test script is set to retrieve the data correctly. Use the appropriate functions in TestComplete to read the data from the external source.

Example (Test Script to Retrieve Data) (Fig 5)

Fig 5

```
function DataDrivenLoginTest()
{
    var dataFile = "C:\\Tests\\TestData.csv";
    var dataTable = DataTable("CSV", dataFile);

    // Loop through each row of the data table
    while (dataTable.Next())
    {
        var username = dataTable.Value("Username");
        var password = dataTable.Value("Password");

        tl-step("Open Application")
        Sys.Process("MyApp").Run();

        tl-step("Enter Username")
        Aliases.MyApp.LoginForm.Username.SetText(username);

        tl-step("Enter Password")
        Aliases.MyApp.LoginForm.Password.SetText(password);

        tl-step("Click Login Button")
        Aliases.MyApp.LoginForm.LoginButton.Click();

        if (Aliases.MyApp.HomePage.Exists)
        {
            Log.Message("Login successful with: " + username);
        }
        else
        {
            Log.Error("Login failed with: " + username);
        }
    }
}
```

© NIMI NOT TO BE REPUBLISHED

6 Run the Data-Driven Test

After adding the data to the external data table and configuring your test script, you can now run the data-driven test. The test will loop through each row of data and execute the steps using the values from the data source.

Adjusting the script with regular information

Objective: At the end of this exercise you shall be able to
- **modify script to use regular test data.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Adjusting the script with regular information in **creating data-driven tests** refers to modifying your test script to work with the actual data, ensuring that your script uses the right data from an external source (like a CSV or Excel file) in a structured and effective way.

Step 1: Set Up Test Data (CSV File)

The first thing you need is the test data. This will be stored in a CSV file. The file could look like this:

TestData.csv (Fig 1)

Fig 1

```
Username,Password
testuser1,password123
testuser2,password456
testuser3,password789
```

This CSV contains three rows of user credentials (one for each test).

Step 2: Initialize Data Table and Load Data

In this step, we will load the CSV data into the DataTable. This allows us to access each username and password as we loop through the file. (Fig 2)

Fig 2

```
function DataDrivenLoginTest()
{
    // Define the path to the CSV file
    var dataFile = "C:\\Tests\\TestData.csv";

    // Load CSV data into a DataTable
    var dataTable = DataTable("CSV", dataFile);
}
```

Explanation

- **DataFile:** Specifies the file path to the CSV file that contains the test data.
- **DataTable:** This object loads the CSV data into memory for further processing.

Step 3: Loop Through Each Row of Data

Once the data is loaded, we will loop through each row to extract the username and password for the test. Each iteration of the loop corresponds to one test. (Fig 3)

Fig 3

```
while (dataTable.Next()) {  // Loop through each row in the CSV
    var username = dataTable.Value("Username");  // Get username from current row
    var password = dataTable.Value("Password");  // Get password from current row
}
```

Explanation

- **DataTable.Next():** Moves to the next row in the CSV.
- **DataTable.Value("Username"):** Retrieves the value of the "Username" column from the current row.
- **DataTable.Value("Password"):** Retrieves the value of the "Password" column from the current row.

Step 4: Open Application

We need to open the application that we are testing. This step will launch the app before performing the login. (Fig 4)

© NIMI NOT TO BE REPUBLISHED

Fig 4

```
tl-step("Open Application");  // Log the action of opening the application
Sys.Process("MyApp").Run();  // Launch the application
```

Expected Output

- **Log Entry:** "Open Application"
- **Action:** The application (MyApp) is started.

Step 5: Enter Username and Password

After launching the application, the script proceeds to enter the username and password for the current row in the DataTable. (Fig 5)

Fig 5

```
tl-step("Enter Username");  // Log the action of entering the username
Aliases.MyApp.LoginForm.Username.SetText(username);  // Set username from DataTable

tl-step("Enter Password");  // Log the action of entering the password
Aliases.MyApp.LoginForm.Password.SetText(password);  // Set password from DataTable
```

Expected Output

- **Log Entry:** "Enter Username" and "Enter Password"
- **Action:** The credentials (username and password) from the current row are typed into the application's login form.

Step 6: Click Login Button

Next, the script simulates clicking the login button on the form to attempt login. (Fig 6)

Fig 6

```
tl-step("Click Login Button");  // Log the action of clicking the login button
Aliases.MyApp.LoginForm.LoginButton.Click();  // Click the login button
```

Expected Output

- **Log Entry:** "Click Login Button"
- **Action:** The login button is clicked in the application.

Step 7: Verify Login Success

After attempting to log in, we need to verify whether the login was successful. This is done by checking if the home page of the application is visible. (Fig 7)

Fig 7

```
if (Aliases.MyApp.HomePage.Exists) {  // Check if HomePage exists
    Log.Message("Login successful with: " + username);  // Log success
} else {  // If HomePage doesn't exist, login failed
    Log.Error("Login failed with: " + username);  // Log failure
}
```

Expected Output

- **If login is successful:** The home page exists, and a success message is logged:
  - **Log Entry:** "Login successful with: testuser1"
- **If login fails:** The home page doesn't exist, and a failure message is logged.
  - **Log Entry:** "Login failed with: testuser2"

© NIMI NOT TO BE REPUBLISHED

Step 8: Repeat for Each User

The script loops through the CSV file, executing the login test for each user.

Example Output (Complete Log)

The output would look something like this. (Fig 8)

Fig 8

```
Step: Open Application
Step: Enter Username
Step: Enter Password
Step: Click Login Button
Login successful with: testuser1

Step: Open Application
Step: Enter Username
Step: Enter Password
Step: Click Login Button
Login failed with: testuser2

Step: Open Application
Step: Enter Username
Step: Enter Password
Step: Click Login Button
Login successful with: testuser3
```

© NIMI NOT TO BE REPUBLISHED

Running the test and analysing result

Objective: At the end of this exercise you shall be able to
- **run data-driven test and analyze results.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Running the test and analyzing results in **Creating Data-Driven Tests** involves executing your data-driven test script, where different sets of input data are used for each iteration, and then reviewing the results to ensure the application behaves as expected.

Step 1: Test Data Setup (CSV)

The first thing you need is the test data. We already know the CSV file (TestData.csv) containing user credentials for login.

TestData.csv

This CSV will be used as input for the data-driven test, where the script will read each row (username and password) and attempt to log in to the application. (Fig 1)

Fig 1

```
Username,Password
testuser1,password123
testuser2,password456
testuser3,password789
```

Step 2: The Test Script

Now, let's look at the test script that runs the login functionality. Here's the step-by-step script. (Fig 2)

Fig 2

```
function DataDrivenLoginTest()
{
    // Define the path to the CSV file
    var dataFile = "C:\\Tests\\TestData.csv";

    // Load CSV data into a DataTable
    var dataTable = DataTable("CSV", dataFile);

    // Loop through each row of the CSV file
    while (dataTable.Next()) {
        var username = dataTable.Value("Username");  // Get username
        var password = dataTable.Value("Password");  // Get password

        tl-step("Open Application");  // Log the action of opening the application
        Sys.Process("MyApp").Run();  // Start the application

        tl-step("Enter Username");  // Log the action of entering the username
        Aliases.MyApp.LoginForm.Username.SetText(username);  // Set username

        tl-step("Enter Password");  // Log the action of entering the password
        Aliases.MyApp.LoginForm.Password.SetText(password);  // Set password

        tl-step("Click Login Button");  // Log the action of clicking the login button
        Aliases.MyApp.LoginForm.LoginButton.Click();  // Click login button

        // Verify if login was successful
        if (Aliases.MyApp.HomePage.Exists) {
            Log.Message("Login successful with: " + username);  // Log success
        } else {
            Log.Error("Login failed with: " + username);  // Log failure
        }
    }
}
```

© NIMI NOT TO BE REPUBLISHED

Step 3: Running the Test

1 Start Test Execution

- Open the test automation tool (e.g., TestComplete).
- Load the project and the script.
- Click the Run button to execute the test.

2 Test Execution Process

- The tool will read the TestData.csv file.
- It will process each row and perform the following steps for each username and password:
  - Open the application.
  - Enter the credentials.
  - Click the login button.
  - Verify if the login was successful by checking if the home page appears.

Step 4: Capturing Test Results

Once the test finishes, the tool will generate a log of execution. The log includes each step performed during the test, and it will indicate whether each test pass or fails.

Test Logs Example (Fig 3)

Fig 3

```
Step: Open Application
Step: Enter Username
Step: Enter Password
Step: Click Login Button
Login successful with: testuser1

Step: Open Application
Step: Enter Username
Step: Enter Password
Step: Click Login Button
Login failed with: testuser2

Step: Open Application
Step: Enter Username
Step: Enter Password
Step: Click Login Button
Login successful with: testuser3
```

Step 5: Analyzing the Results

Now let's analyze the output based on the example logs above.

Breakdown of Results

1 User: testuser1

- The test steps for testuser1 were successful. The log indicates.
  - **Login successful:** The home page was found, indicating that the login attempt was successful.
  - **Log Entry:** "Login successful with: testuser1"

2 User: testuser2

- The test for testuser2 failed because the home page didn't appear. This suggests a login failure.
  - **Login failed:** This could be due to incorrect credentials, a server issue, or other reasons.
  - **Log Entry:** "Login failed with: testuser2"

3 User: testuser3

- The test for testuser3 was successful as the home page was found.
  - **Login successful:** The login for this user was successful.
  - **Log Entry:** "Login successful with: testuser3"

Test Execution Summary

- **Total Number of Users Tested:** 3
- **Successful Logins:** 2 (testuser1, testuser3)
- **Failed Logins:** 1 (testuser2)

Step 6: Test Summary and Report

After the test finishes, TestComplete or your test automation tool will generate a Test Report. The report includes the summary of each test step and shows whether it passed or failed.

Test Summary Report Example (Fig 4)

Fig 4

```
Test Name: DataDrivenLoginTest
------------------------------
Total Steps: 12
Passed Steps: 10
Failed Steps: 2

User Test Results:
------------------
- testuser1: Login successful
- testuser2: Login failed
- testuser3: Login successful

Overall Result: Failed (1 test failed)
```

Important Points

1 Successful Tests

- The login was successful for testuser1 and testuser3.
- These steps will be marked as green in the log to indicate success.

© NIMI NOT TO BE REPUBLISHED

2 **Failed Test**

- The login failed for testuser2.
- This step will be marked with a red X in the log to indicate failure.

Analyzing the Failure for testuser2

- The failure could be due to various reasons:
- **Incorrect Credentials:** The password for testuser2 might be wrong.
- **Network Issues:** There might have been a delay in loading the home page.
- **UI Changes:** The login page might have changed, causing the tool to fail finding the home page element.

In this case, you can

- Double-check the credentials in TestData.csv.
- Manually test the login for testuser2 to verify the credentials.
- Inspect the application's login functionality to see if any UI changes could have impacted the test.

Step 7: Fixing Issues and Re-running the Test (If Needed)

If you encounter failures, you can

1 **Fix the issue** (for example, correct the credentials in the CSV, or fix the application's login issue).

2 **Re-run the test** to ensure everything works correctly.

3 Repeat the **Test Execution** and check the **new logs**.

Final Output Example Summary

After running the test for the three users, the output might look like this in the log and report. (Fig 5)

Fig 5

```
Test Execution Summary:
-----------------------
- Test: DataDrivenLoginTest
- Total Users: 3
- Passed: 2
- Failed: 1
- Execution Time: 2 minutes

Step-by-Step Log:
-----------------
[OK] Open Application (Step 1)
[OK] Enter Username: testuser1 (Step 2)
[OK] Enter Password: password123 (Step 3)
[OK] Click Login Button (Step 4)
[OK] Login successful with: testuser1 (Step 5)

[OK] Open Application (Step 1)
[OK] Enter Username: testuser2 (Step 2)
[OK] Enter Password: password456 (Step 3)
[OK] Click Login Button (Step 4)
[ERROR] Login failed with: testuser2 (Step 5)

[OK] Open Application (Step 1)
[OK] Enter Username: testuser3 (Step 2)
[OK] Enter Password: password789 (Step 3)
[OK] Click Login Button (Step 4)
[OK] Login successful with: testuser3 (Step 5)
```

© NIMI NOT TO BE REPUBLISHED

Data driven testing tips

Objective: At the end of this exercise you shall be able to
- **use diverse data and ensure test consistency.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Data-driven testing tips in Creating Data-Driven Tests help improve the effectiveness and efficiency of your test automation by ensuring that tests are comprehensive, reliable, and easy to maintain.

Step 1: Identify Test Scenarios

Before starting with data-driven testing, identify test cases that require different input values to ensure comprehensive validation of your system.

Example Test Scenario

Let's say you're testing a Login Functionality where the input data includes usernames and passwords.

Example Test Case

- **Scenario:** Verify the login functionality works with valid and invalid credentials.
- **Data:** Different sets of usernames and passwords.

Step 2: Prepare Your Test Data

Prepare your data in a structured format. This could be in the form of:

- CSV files
- Excel files
- JSON files
- Database entries
- Hardcoded data in the test code

Sample Test Data (CSV Format)

Here's an example of what your test data might look like in CSV format. (Fig 1)

Fig 1

Username	Password	Expected Result
user1	pass1	Login Successful
user2	pass2	Login Successful
user3	wrongpass	Login Failed
invaliduser	pass4	Login Failed
""	""	Login Failed

Step 3: Choose a Testing Framework

Choose a framework that supports data-driven testing. Common frameworks include

- **JUnit** (Java)
- **TestNG** (Java)
- **PyTest** (Python)
- **NUnit** (.NET)

In this example, we'll use **JUnit 5** for Java.

Step 4: Implement the Test

Write a test method that will take the test data and use it to validate the functionality.

© NIMI NOT TO BE REPUBLISHED

Example Code (JUnit 5) (Fig 2)

Fig 2

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class LoginTest {

    @ParameterizedTest
    @CsvSource({
        "user1, pass1, Login Successful",
        "user2, pass2, Login Successful",
        "user3, wrongpass, Login Failed",
        "invaliduser, pass4, Login Failed",
        "'', '', Login Failed"
    })
    void testLogin(String username, String password, String expectedResult) {
        // Simulate a login function
        String result = login(username, password);
        assertEquals(expectedResult, result);
    }

    // Mock login method for simplicity
    String login(String username, String password) {
        if (username.isEmpty() || password.isEmpty()) {
            return "Login Failed";
        }
        if (("user1".equals(username) && "pass1".equals(password)) ||
            ("user2".equals(username) && "pass2".equals(password))) {
            return "Login Successful";
        }
        return "Login Failed";
    }
}
```

© NIMI NOT TO BE REPUBLISHED

Step 5: Run the Tests

When you run the tests, the testing framework will automatically use the data provided in the @CsvSource and execute the testLogin method for each dataset.

Expected Output

The test results show that each test case is executed with the provided parameters, and it will show whether the test passed or failed. In this case, the tests pass for all combinations of inputs, verifying that the system behaves as expected. (Fig 3)

Fig 3

```
LoginTest > testLogin[user1, pass1, Login Successful] PASSED
LoginTest > testLogin[user2, pass2, Login Successful] PASSED
LoginTest > testLogin[user3, wrongpass, Login Failed] PASSED
LoginTest > testLogin[invaliduser, pass4, Login Failed] PASSED
LoginTest > testLogin[, , Login Failed] PASSED
```

Step 6: Review and Analyze Test Results

Once the tests are complete, analyze the results. Here's what you should look for:

- **Pass/Fail Status:** Ensure the tests either pass or fail based on the expected outcomes.
- **Error Messages:** If a test fails, check the error message to understand what went wrong.
- **Data Coverage:** Ensure that your test data covers a wide range of scenarios (valid, invalid, edge cases).

Step 7: Extend with More Data Sets (Optional)

To increase test coverage, you can add more test data. For example, add cases like empty usernames, long passwords, or special characters to see how the system reacts.

Example (Fig 4)

Fig 4

```
@CsvSource({
    "user1, pass1, Login Successful",
    "user2, pass2, Login Successful",
    "user3, wrongpass, Login Failed",
    "invaliduser, pass4, Login Failed",
    "'', '', Login Failed",
    "longuser, longpassword123, Login Successful",
    "user@name, special@pass, Login Failed"
})
```

This extends your test coverage to handle even more real-world scenarios.

Step 8: Optional: Parameterize Data From External Sources

If your data is large or coming from external sources like a database or an Excel file, you can parameterize your test method to fetch data from these sources dynamically.

For example, in JUnit

- Use an @MethodSource annotation to load data from a method that returns a Stream or Iterable of test data.
- Or use @ValueSource for single values.

Step 9: Execute and Analyze Again

Once the test has been updated with more data, rerun the tests to check if all data sets pass correctly.

Tips for Data-Driven Testing

1. **Use meaningful test data:** Choose data that simulates real-world use cases, including edge cases, boundary values, and invalid inputs.
2. **Automate test data:** Consider storing test data externally (e.g., in CSV, Excel, or database) so that it's easier to maintain and update.
3. **Limit the number of tests:** Avoid excessive test data that might slow down the test execution unnecessarily.
4. **Verify expected results:** Ensure the expected results are correctly defined based on the test data (e.g., for login: success or failure).
5. **Handle data dependencies:** If tests depend on data in a certain order, consider using setup and teardown methods to prepare and clean up the environment.

© NIMI NOT TO BE REPUBLISHED

How do you read text from an application?

Objective: At the end of this exercise you shall be able to
- **extract text from application using automation tools.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

To **read text from an application** during automated testing, you use specific functions and objects provided by the automation tool (e.g., TestComplete, Selenium, or others) that can interact with the graphical user interface (GUI) of the application and extract text from various elements like labels, buttons, or text boxes.

Here are the general steps for reading text from an application:

1 Identify the Object Containing the Text

- **Locate the Element:** Use the automation tool to identify the UI element from which you want to extract text. This could be a label, text field, button, or any other element that contains visible text.
- **Object Mapping:** The automation tool must be able to map the object (using properties such as ID, name, or class) to interact with it.

For example, in **TestComplete,** you may use the object's Text property to retrieve the text.

2 Use the Automation Tool's Text Reading Methods:

Depending on the tool you're using, there are specific methods to get the text content of an element.

TestComplete (Example) (Fig 1)

Fig 1

```
var text = Aliases.MyApp.MainWindow.Label.Text;
Log.Message("The text in the label is: " + text);
```

- **Step-by-Step**
  - **Aliases:** Represents the application being tested.
  - **MainWindow:** The main window of the application.
  - **Label:** The object (like a label) containing the text you want to read.
  - **Text:** The property that holds the text.

Selenium (Example with Java) (Fig 2)

Fig 2

```
WebElement element = driver.findElement(By.id("labelID"));
String text = element.getText();
System.out.println("The text is: " + text);
```

- **Driver.findElement():** Finds the element on the page by a unique locator (like By.id).
- **GetText():** Retrieves the text content from the located element.

3 Handle Dynamic Text (If Needed)

- **Wait for Text:** If the text is dynamic (changes after user interaction), ensure that your test waits for the text to appear or update before reading it.
- **Synchronization:** Use synchronization methods (like Wait in TestComplete or WebDriverWait in Selenium) to wait for elements to be present or updated before reading their text.

© NIMI NOT TO BE REPUBLISHED

TestComplete (Waiting for Text) (Fig 3)

Fig 3

```
var label = Aliases.MyApp.MainWindow.Label;
label.WaitProperty("Text", "Expected Text", 5000); // Wait for 5 seconds
var text = label.Text;
Log.Message("The expected text is: " + text);
```

4 Extracting Text from Multiple Elements

- If you need to read text from multiple elements (like all items in a list), you can loop through the elements and extract text from each.

TestComplete (Loop Through Multiple Elements) (Fig 4)

Fig 4

```
var labels = Aliases.MyApp.MainWindow.FindChildren([cmoLabel]);
for (var i = 0; i < labels.length; i++) {
    Log.Message("Label text: " + labels[i].Text);
}
```

5 Handling Special Text Formats (Rich Text, Dynamic Content)

- If the text includes rich formatting (bold, italic, etc.), or if the text is dynamically generated or loaded, additional steps may be required to extract all content or deal with updates in real-time.

For Rich Text: Some tools may provide specific methods to extract styled or formatted text from rich text controls or browsers.

For Dynamic Content: Always ensure synchronization and proper handling of dynamic text (for instance, with Wait methods).

6 Error Handling

- Make sure to handle errors gracefully when text cannot be found, or when elements are not available. This can be done through error handling mechanisms provided by the tool.
- For example, in TestComplete, you can use try-catch blocks, and in Selenium, you can use try-catch to catch NoSuchElementException.

Example Summary

In TestComplete (Fig 5)

Fig 5

```
try {
   var labelText = Aliases.MyApp.MainWindow.Label.Text;
   Log.Message("Text read: " + labelText);
} catch (e) {
   Log.Error("Failed to read text: " + e.message);
}
```

In Selenium (Fig 6)

Fig 6

```
try {
   WebElement element = driver.findElement(By.id("labelID"));
   String labelText = element.getText();
   System.out.println("Text read: " + labelText);
} catch (NoSuchElementException e) {
   System.out.println("Element not found: " + e.getMessage());
}
```

© NIMI NOT TO BE REPUBLISHED

Reading text from an application

Objective: At the end of this exercise you shall be able to
- **extract text from application using automation methods.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Reading text from an application involves extracting the content displayed on UI elements (such as labels, buttons, text fields, etc.) using automation tools. The process typically includes identifying the relevant UI elements and using specific functions or methods provided by the automation tool to retrieve the text.

Steps for Reading Text from an Application

1 Identify the UI Element Containing the Text

- First, you need to locate the UI element (such as a label, button, text box, etc.) that contains the text you want to read.
- The element can be located by its properties (e.g., ID, name, class).

2 Use Automation Tool's Methods to Extract the Text: Different automation tools provide methods to interact with UI elements and extract the text.

- **TestComplete Example (Fig 1)**

Fig 1

```
var text = Aliases.MyApp.MainWindow.Label.Text;
Log.Message("The text in the label is: " + text);
```

- **Selenium Example (Java) (Fig 2)**

Fig 2

```
WebElement element = driver.findElement(By.id("labelID"));
String text = element.getText();
System.out.println("The text is: " + text);
```

3 Handle Dynamic Content

- If the text is dynamic (i.e., it changes over time), ensure that you use wait mechanisms to allow the text to load or update before reading it.
- **TestComplete (Fig 3)**

Fig 3

```
var label = Aliases.MyApp.MainWindow.Label;
label.WaitProperty("Text", "Expected Text", 5000); // Wait for text to change
var text = label.Text;
Log.Message("Text after waiting: " + text);
```

© NIMI NOT TO BE REPUBLISHED

4 Extract Text from Multiple Elements

- To read text from multiple elements (e.g., a list of labels), loop through the elements and extract their text.
- **TestComplete (Fig 4)**

Fig 4

```
var labels = Aliases.MyApp.MainWindow.FindChildren([cmoLabel]);
for (var i = 0; i < labels.length; i++) {
    Log.Message("Label text: " + labels[i].Text);
}
```

- **Selenium (Fig 5)**

Fig 5

```
List<WebElement> labels = driver.findElements(By.className("labelClass"));
for (WebElement label : labels) {
    System.out.println("Label text: " + label.getText());
}
```

5 Handle Errors Gracefully

- Always ensure that your script can handle situations where the text might not be found or the element might not be visible. This prevents your test from failing unexpectedly.
- **TestComplete (Fig 6)**

Fig 6

```
try {
    var labelText = Aliases.MyApp.MainWindow.Label.Text;
    Log.Message("Text: " + labelText);
} catch (e) {
    Log.Error("Error reading text: " + e.message);
}
```

- **Selenium (Java) (Fig 7)**

Fig 7

```
try {
    WebElement element = driver.findElement(By.id("labelID"));
    String labelText = element.getText();
    System.out.println("Text: " + labelText);
} catch (NoSuchElementException e) {
    System.out.println("Element not found: " + e.getMessage());
}
```

© TNMI NOT TO BE REPUBLISHED

Teaching fonts to win runner

Objective: At the end of this exercise you shall be able to
- **teach WinRunner to recognize and read fonts.**

Requirements	
Tool/Instruments	
• A working PC	- as reqd.

PROCEDURE

Step 1: Understand WinRunner Basics

WinRunner automates GUI testing by recording user actions and generating scripts that simulate them. It works with applications to perform testing tasks, such as clicking buttons or entering text.

Step 2: Set Up WinRunner for Font Testing

1 **Install WinRunner:** Ensure WinRunner is installed on your system.

2 **Configure the Application:** Identify the application where you need to test fonts (e.g., text editors, websites displaying different fonts).

3 **Enable Window and Text Recognition:** Make sure WinRunner is set up to recognize window elements, text areas, and any specific text you want to evaluate.

Step 3: Record Actions in WinRunner

1 **Start Recording:** Open WinRunner and start recording your actions (e.g., opening an application, navigating to a text section).

2 **Perform Actions:** Manually interact with the application displaying text in different fonts. For example:
- Open a document in a word processor.
- Type text in different fonts (e.g., Arial, Times New Roman, Verdana).
- Observe how the text is displayed and verify the font.

Step 4: Add Font Checkpoints in Scripts

1 **Create Checkpoints:** After recording the actions, add checkpoints to verify the font properties of the displayed text. WinRunner uses a specific syntax to check various GUI elements.

2 **Font Property Validation:** You can use the text_get_font function to capture the font used by a particular text element in your application.

For example (Fig 1)

Fig 1

```
font = text_get_font("window_name", "text_area_name");
if (font == "Arial") {
    print("Font is Arial.");
} else {
    print("Font is not Arial.");
}
```

This script checks the font of the text in a specific area and prints a message based on whether the font is Arial.

Step 5: Extend with Additional Tests

1 **Test Multiple Fonts:** Create additional test steps for different fonts.
- Input text with fonts like Verdana, Times New Roman, etc.
- For each test case, use WinRunner to capture and validate the font.

© NIMI NOT TO BE REPUBLISHED

2 **Font Size and Style:** Check for different styles (bold, italic) and font sizes, adjusting your script to capture these properties as well.

Example of checking both font style and size (Fig 2)

Fig 2

```
font = text_get_font("window_name", "text_area_name");
size = text_get_font_size("window_name", "text_area_name");

if (font == "Arial" && size == 12) {
    print("Font is Arial, size is 12.");
} else {
    print("Font or size is different.");
}
```

Step 6: Run the Tests

Once your test scripts are ready, execute them

1 **Run Scripts:** Execute the scripts using WinRunner.

2 **Check Output:** Review the output logs to ensure that fonts and other text properties are recognized correctly.

Step 7: Interpret Test Results

1 **Validation:** If the expected font (e.g., Arial) is displayed correctly in the test case, the script will output "Font is Arial". If the font is not what was expected, it will print a failure message.

2 **Debug:** If the test fails (e.g., font mismatch), investigate why the font wasn't recognized and refine your test scripts accordingly.

Step 8: Automate Font Testing for Other Applications

You can apply this approach to other applications that require font testing, such as web pages, text processors, or any GUI that uses varying fonts. The steps to record actions, check font properties, and validate them remain similar across different applications.

Output Example

1 **Font Test Passed**
- "Font is Arial"
- "Font is Verdana"
- "Font is Times New Roman"

2 **Font Test Failed**
- "Font is not Arial"
- "Font is not Times New Roman"

© NIMI NOT TO BE REPUBLISHED

Verifying text

Objective: At the end of this exercise you shall be able to
- **verify text content in application using automation.**

Requirements

Tool/Instruments

• A working PC	- as reqd.

PROCEDURE

Verifying text in an application involves checking that the text displayed within a UI element matches the expected value. This is essential in automated testing to ensure that the correct content is shown at the right place in the application.

Here's how you can verify text in various automation tools:

Steps to Verify Text

1 Identify the Element Containing the Text

- The first step is to locate the element that holds the text you want to verify. This could be a label, button, text box, etc.
- Elements are usually identified by properties like ID, class name, XPath, or other attributes.

2 Extract the Text from the Element

- Use specific functions or methods to extract the text content from the element.
- Ensure the text is extracted accurately, especially when dealing with dynamic or formatted text.

3 Compare the Extracted Text with the Expected Value

- Once the text is extracted, compare it to the expected value, either using an assertion or a conditional check.

Example in Different Automation Tools

TestComplete (Fig 1)

Fig 1

```
var actualText = Aliases.MyApp.MainWindow.Label.Text; // Get text from the label
var expectedText = "Expected Text"; // Define expected text

if (actualText == expectedText) {
    Log.Message("Text verification passed!");
} else {
    Log.Error("Text verification failed! Expected: " + expectedText + ", but got: "
}
```

TestComplete allows you to extract text from an object and compare it to the expected value.

- **Aliases.MyApp.MainWindow.Label.Text:** Accesses the text property of the label.
- **Text Comparison:** Compares the actual text with the expected text.

Selenium (Java) (Fig 2)

Fig 2

```
WebElement element = driver.findElement(By.id("labelID"));
String actualText = element.getText();
String expectedText = "Expected Text";

if (actualText.equals(expectedText)) {
    System.out.println("Text verification passed!");
} else {
    System.out.println("Text verification failed! Expected: " + expectedText + "
}
```

© NIMI NOT TO BE REPUBLISHED

In Selenium, you can extract text from an element and assert that it matches the expected value.

- driver.findElement(By.id("labelID")): Locates the element by its ID.
- getText(): Retrieves the text content of the element.
- equals(): Compares the actual and expected text values.

WinRunner (Fig 3)

Fig 3

```
text_check("Expected Text", 1);
```

WinRunner can verify text using the text_check() function, which checks if the text appears in the specified GUI element.

- text_check(): Verifies the text in the specified object.
- 1: The index of the object to be checked.

Handling Dynamic Text

If the text is dynamic (i.e., it changes based on actions or is loaded asynchronously), ensure synchronization before verifying.

- In TestComplete: Use Wait or WaitProperty to wait for the text to appear.
- In Selenium: Use WebDriverWait to wait for the element to become visible before extracting and verifying the text.

Example of Waiting for Text

TestComplete (Fig 4)

Fig 4

```
var label = Aliases.MyApp.MainWindow.Label;
label.WaitProperty("Text", "Expected Text", 5000); // Wait for up to 5 seconds
var actualText = label.Text;
Log.Message("Actual text: " + actualText);
```

Selenium (Fig 5)

Fig 5

```
WebDriverWait wait = new WebDriverWait(driver, 10);
WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("lab
String actualText = element.getText();
System.out.println("Actual text: " + actualText);
```

© NIMI NOT TO BE REPUBLISHED

Running the test on a new version

Objective: At the end of this exercise you shall be able to
- **run text verification tests on application updates.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Running the test on a new version involves executing a previously created test script on a new version of the application to verify that the text displayed in the user interface (UI) still matches the expected values. This process ensures that updates or changes to the application have not unintentionally affected the displayed text.

Step-by-Step Guide to Running the Test on a New Version

Step 1: Define Test Objectives

- What is being tested? (e.g., performance, accuracy, stability)
- What changes exist in the new version?
- What metrics will determine success?

Step 2: Set Up the Environment

- Install or deploy the new version.
- Ensure dependencies and configurations are correct.
- Example (if using a Python environment) (Fig 1)

Fig 1

```
pip install -r requirements.txt
```

Step 3: Prepare Test Cases

- Identify test inputs (unit tests, functional tests, regression tests).
- Example test cases:
  - Basic functionality test
  - Edge case handling
  - Performance under load

Step 4: Execute Tests and Capture Output

Example 1: Running Automated Tests in Python (Fig 2)

Fig 2

```
import unittest
from my_module import my_function  # Replace with actual module

class TestNewVersion(unittest.TestCase):
    def test_case_1(self):
        input_data = "Hello, world!"
        expected_output = "HELLO, WORLD!"  # Assuming new version converts to uppercase
        result = my_function(input_data)
        self.assertEqual(result, expected_output)

if __name__ == '__main__':
    unittest.main()
```

© NIMI NOT TO BE REPUBLISHED

Expected Output (Terminal) (Fig 3)

Fig 3

```
Running test_case_1...
OK
```

Example 2: Running Performance Test (Fig 4)

Fig 4

```
import time
start_time = time.time()
output = my_function("Performance Test")
end_time = time.time()

print(f"Execution Time: {end_time - start_time} seconds")
```

Expected Output (Fig 5)

Fig 5

```
Execution Time: 0.0021 seconds
```

Step 5: Compare Results (Old vs. New Version) (Fig 6)

Fig 6

Test Case	Old Version Output	New Version Output	Pass/Fail
Case 1	"hello, world!"	"HELLO, WORLD!"	✅ Pass
Performance	0.005s	0.002s	✅ Improved

Step 6: Log Issues and Fixes

- Record any errors or failures.
- Example error message (Fig 7)

Fig 7

```
ERROR: Expected 'HELLO, WORLD!' but got 'hello, world!'
```

Step 7: Generate Test Report

- Compile results into a summary report.
- Example (Fig 8)

Fig 8

```
pytest --html=report.html
```

© NIMI NOT TO BE REPUBLISHED

Text checkpoint tips

Objective: At the end of this exercise you shall be able to
- **ensure accurate text verification with checkpoint techniques.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Text checkpoint tips help ensure that the text displayed in an application matches the expected content during automated testing. These checkpoints are essential for verifying the accuracy of UI text and preventing regressions after updates or changes to the application.

Step 1: Define Checkpoints in the Process

Checkpoints should be placed at key stages of text processing. Example stages:

1 **Raw Input** → Before processing

2 **Preprocessing** → Cleaning and tokenization

3 **Model Output** → After prediction

4 **Post-processing** → Formatting results

5 **Final Output** → Ready-to-use text

Step 2: Implement Checkpoints in Code

Use logging or print statements to capture outputs at each stage.

Example: Text Processing Pipeline (Python) (Fig 1)

Fig 1

```python
import logging

# Setup logging for checkpoints
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")

def text_pipeline(input_text):
    logging.info(f"Checkpoint 1 - Raw Input: {input_text}")

    # Step 1: Preprocessing (Lowercasing and removing punctuation)
    processed_text = input_text.lower().replace(".", "").replace(",", "")
    logging.info(f"Checkpoint 2 - Preprocessed Text: {processed_text}")

    # Step 2: Model Simulation (Reversing text as an example)
    model_output = processed_text[::-1]
    logging.info(f"Checkpoint 3 - Model Output: {model_output}")

    # Step 3: Post-processing (Capitalizing first letter)
    final_output = model_output.capitalize()
    logging.info(f"Checkpoint 4 - Final Output: {final_output}")

    return final_output

# Run the pipeline
output = text_pipeline("Hello, World!")
print("Final Output:", output)
```

© NIMI NOT TO BE REPUBLISHED

Step 3: Capture Checkpoint Outputs

Expected Console Output (Fig 2)

Fig 2

```
2025-02-10 12:00:00 - INFO - Checkpoint 1 - Raw Input: Hello, World!
2025-02-10 12:00:00 - INFO - Checkpoint 2 - Preprocessed Text: hello world
2025-02-10 12:00:00 - INFO - Checkpoint 3 - Model Output: dlrow olleh
2025-02-10 12:00:00 - INFO - Checkpoint 4 - Final Output: Dlrow olleh
Final Output: Dlrow olleh
```

Step 4: Compare Results at Each Checkpoint (Fig 3)

Fig 3

Checkpoint	Text Output
Raw Input	"Hello, World!"
Preprocessed	"hello world"
Model Output	"dlrow olleh"
Final Output	"Dlrow olleh"

Step 5: Debugging with Checkpoints

If the output is incorrect, check intermediate checkpoints to find where the issue occurred.

✓ **Example Fix:** If the model output is wrong, adjust the preprocessing step.

✕ **Issue:** If punctuation is missing but shouldn't be, fix the preprocessing function.

Step 6: Automate Checkpoint Logging

For large-scale testing, log checkpoints to a file (Fig 4)

This stores all steps for later debugging.

Fig 4

```
logging.basicConfig(filename="checkpoints.log", level=logging.INFO)
```

© NIMI NOT TO BE REPUBLISHED

What is a batch test?

Objective: At the end of this exercise you shall be able to
- **run multiple tests simultaneously for efficient execution.**

Requirements	
Tool/Instruments	
• A working PC	- as reqd.

PROCEDURE

What is a Batch Test?

A **batch test** is a process of running multiple test cases automatically in a single execution. Instead of testing one input at a time, batch testing processes a **set of inputs** to check performance, correctness, and consistency.

Why Use Batch Testing?

- Saves time by testing multiple cases at once
- Ensures consistency across various inputs
- Detects failures in large-scale testing
- Helps compare results between different system versions

Step-by-Step Guide for Running a Batch Test

Step 1: Define Test Cases

Prepare a batch of test inputs and expected outputs.

Example Test Cases for a Text Processing System (Fig 1)

Fig 1

Test Case	Input Text	Expected Output
1	"Hello, World!"	"hello world"
2	"This is a TEST."	"this is a test"
3	"123 ABC!"	"123 abc"

© NIMI NOT TO BE REPUBLISHED

Step 2: Implement Batch Testing in Code

Here's a **Python script** that runs batch tests on a text processing function. (Fig 2)

Fig 2

```python
import logging

# Setup logging for tracking results
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")

# Sample text processing function (Lowercase and remove punctuation)
def text_processing(input_text):
    return input_text.lower().replace("!", "").replace(".", "")

# Define batch test cases (input → expected output)
test_cases = [
    ("Hello, World!", "hello world"),
    ("This is a TEST.", "this is a test"),
    ("123 ABC!", "123 abc"),
]

# Run batch tests
def batch_test():
    passed = 0
    failed = 0

    for i, (input_text, expected_output) in enumerate(test_cases, start=1):
        result = text_processing(input_text)
        if result == expected_output:
            logging.info(f"Test Case {i}: ✅ Pass | Input: {input_text} | Output: {result}")
            passed += 1
        else:
            logging.error(f"Test Case {i}: ❌ Fail | Input: {input_text} | Expected: {expected_output} | Got: {result}")
            failed += 1

    # Summary Report
    logging.info(f"\nBatch Test Summary: {passed} Passed, {failed} Failed")

# Run the batch test
batch_test()
```

© NIMI NOT TO BE REPUBLISHED

Step 3: Capture Batch Test Output

- Expected Console Output (Fig 3)

Fig 3

```
2025-02-10 12:00:00 - INFO - Test Case 1: ✅ Pass | Input: Hello, World! | Output: hello world
2025-02-10 12:00:00 - INFO - Test Case 2: ✅ Pass | Input: This is a TEST. | Output: this is a
2025-02-10 12:00:00 - INFO - Test Case 3: ✅ Pass | Input: 123 ABC! | Output: 123 abc

2025-02-10 12:00:00 - INFO - Batch Test Summary: 3 Passed, 0 Failed
```

Step 4: Analyze Batch Test Results

- ✓ If all tests pass, the function works correctly for the given cases.
- ✗ If a test fails, debug the incorrect output by checking intermediate steps.
- Store results in a file using (Fig 4)

Fig 4

```python
logging.basicConfig(filename="batch_test_results.log", level=logging.INFO)
```

Step 5: Automate and Extend Batch Testing

- **For larger datasets,** store test cases in a CSV file and read dynamically.
- **For performance testing,** measure execution time. (Fig 5)
- **For machine learning models,** batch test different inputs to evaluate accuracy.

Fig 5

```
import time
start_time = time.time()
batch_test()
end_time = time.time()
print(f"Total Execution Time: {end_time - start_time} seconds")
```

© NIMI
NOT TO BE REPUBLISHED

Programming a batch test

Objective: At the end of this exercise you shall be able to
- **automate multiple tests in a single execution**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Step-by-Step Guide to Programming a Batch Test

A **batch test** runs multiple test cases automatically in a single execution. Below, we'll implement a batch test in Python, which processes multiple inputs, compares expected vs. actual outputs, and logs results.

Step 1: Define the Test Function

First, we define a function to be tested. Here's an example function that **processes text** (converts to lowercase and removes punctuation). (Fig 1)

Fig 1

```
import re

def text_processing(input_text):
    """Function to preprocess text: lowercase and remove punctuation."""
    return re.sub(r'[^\w\s]', '', input_text).lower()
```

© NIMI NOT TO BE REPUBLISHED

Step 2: Create Batch Test Cases

We define multiple test inputs with expected outputs. (Fig 2)

Fig 2

```
test_cases = [
    ("Hello, World!", "hello world"),
    ("This is a TEST.", "this is a test"),
    ("123 ABC!", "123 abc"),
    ("Python 3.9, is great!", "python 39 is great"),
    ("$1000 in cash", "1000 in cash")
]
```

Each test consists of:

- **Input text** → Sent to the function
- **Expected output** → What we expect the function to return

Step 3: Write the Batch Testing Script

We run the test cases one by one, compare results, and track successes/failures. (Fig 3)

Fig 3

```
import logging

# Configure logging to track test results
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")

def batch_test():
    passed = 0
    failed = 0

    for i, (input_text, expected_output) in enumerate(test_cases, start=1):
        result = text_processing(input_text)

        if result == expected_output:
            logging.info(f"Test Case {i}: ✅ Pass | Input: {input_text} | Output: {result}")
            passed += 1
        else:
            logging.error(f"Test Case {i}: ❌ Fail | Input: {input_text} | Expected: {expected_output} | Got: {result}")
            failed += 1

    # Summary Report
    logging.info(f"\nBatch Test Summary: {passed} Passed, {failed} Failed")

# Run the batch test
batch_test()
```

Step 4: Expected Output

When the script runs, it logs results for each test case and a final summary.

✓ Example Console Output (Fig 4)

Fig 4

```
2025-02-10 12:00:00 - INFO - Test Case 1: ✅Pass | Input: Hello, World! | Output: hello world
2025-02-10 12:00:00 - INFO - Test Case 2: ✅Pass | Input: This is a TEST. | Output: this is a test
2025-02-10 12:00:00 - INFO - Test Case 3: ✅Pass | Input: 123 ABC! | Output: 123 abc
2025-02-10 12:00:00 - INFO - Test Case 4: ❌Fail | Input: Python 3.9, is great! | Expected: python 39 is
great | Got: python 39 is great
2025-02-10 12:00:00 - INFO - Test Case 5: ✅Pass | Input: $1000 in cash | Output: 1000 in cash
```

Step 5: Improving the Batch Test

- Store results in a file. (Fig 5)

Fig 5

```
logging.basicConfig(filename="batch_test_results.log", level=logging.INFO)
```

- Measure exec ution time. (Fig 6)

Fig 6

```
import time
start_time = time.time()
batch_test()
end_time = time.time()
print(f"Total Execution Time: {end_time - start_time:.4f} seconds")
```

- Use a CSV file for test cases. (Fig 7)

Fig 7

```
import csv

with open("test_cases.csv", newline="") as file:
    reader = csv.reader(file)
    test_cases = list(reader)
```

© TNSCERT NOT TO BE REPUBLISHED

Running the batch test on version IB

Objective: At the end of this exercise you shall be able to
- **execute batch tests on version IB efficiently.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Running a batch test on **version IB** involves executing a series of tests across a specific version of your application, in this case, version IB. This process typically occurs after the application has undergone an update or when a new version is deployed, ensuring that the application behaves as expected after the changes.

Here's how you can go about running batch tests on **version IB**:

1 Set Up the Testing Environment for Version IB

- Ensure that version IB of the application is deployed to the test environment or is accessible for testing.
- Verify that all necessary components, such as databases, third-party services, or dependencies, are properly configured for the new version.

2 Prepare the Batch Test Suite

- Create a batch test suite that includes all the individual test cases you want to run against version IB. This can include functional, regression, performance, and sanity tests.
- Define the tests either through a Test Suite file (such as TestNG XML or JUnit in Java) or by scripting in a testing tool like TestComplete or Selenium.

3 Set Up Test Configuration for Version IB

- If you're using a tool like Selenium, you can specify the URL for version IB, or you may need to configure a build-specific environment for your tests.
- If you are using TestComplete or WinRunner, you can configure the test to point to the new version or use version-specific configurations.

4 Run the Batch Test

- **In TestComplete:** Create a Test Suite and add your test scripts, specifying the version IB environment to run the tests on.
  - Example in JavaScript (TestComplete) (Fig 1)

Fig 1

```
function RunBatchTestsForVersionIB() {
  var testScripts = ["Test1", "Test2", "Test3"];
  for (var i = 0; i < testScripts.length; i++) {
    Log.Message("Running test on Version IB: " + testScripts[i]);
    TestedApps[testScripts[i]].Run();
  }
}
```

© NIMI NOT TO BE REPUBLISHED

- **In Selenium (TestNG):** Define the test suite and execute the tests on the version IB application.
  - Example TestNG XML configuration (Fig 2)

Fig 2

```
<suite name="Batch Test Suite for Version IB" parallel="tests" thread-count="3">
  <test name="Regression Test for Version IB">
    <parameter name="version" value="IB"/>
    <classes>
      <class name="Test1"/>
      <class name="Test2"/>
    </classes>
  </test>
</suite>
```

5 Monitor and Analyze Test Results

- Once the batch tests are executed, monitor the results. If you're using **TestComplete**, you can view the results in the **Test Log** or **Report**.
- **Selenium with TestNG** will provide a detailed **TestNG report** showing the success or failure of each individual test case in the batch.

6 Handle Issues and Report

- If there are test failures, analyze the logs and reports to determine the root cause. This could involve investigating the test scripts, the version IB of the application, or the test environment.
- Create a bug report or pass the issue to the development team if necessary.

7 Iterate for Further Versions

- After running the tests for version IB, you may need to modify or update the tests based on the results or any application changes in future versions.

Example Process in Selenium with TestNG for Version IB

Assume your **version IB** of the app is available at http://yourapp-version-IB.com. The batch tests will be set up as follows:

1 Test Class (Test1.java) (Fig 3)

Fig 3

```
@Test
public void testMethod1() {
    WebDriver driver = new ChromeDriver();
    driver.get("http://yourapp-version-IB.com");
    // Your test logic here
}
```

2 TestNG XML Configuration for Version IB (Fig 4)

Fig 4

```
<suite name="Batch Test Suite for Version IB">
  <test name="Version IB Tests">
    <parameter name="version" value="IB"/>
    <classes>
      <class name="Test1"/>
      <class name="Test2"/>
    </classes>
  </test>
</suite>
```

3 Running the Tests: You would execute the TestNG suite on version IB with the following command (assuming Maven is configured). (Fig 5)

Fig 5

```
mvn test -DsuiteXmlFile=batch-test-suite.xml
```

© NIMI NOT TO BE REPUBLISHED

Analysing the batch test results

Objective: At the end of this exercise you shall be able to
- **execute batch tests on version IB efficiently.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Step-by-Step Guide to Analyzing Batch Test Results

After running a batch test, analyzing the results is crucial to identify failures, track performance, and improve the system. Below is a structured approach to analyzing batch test results.

Step 1: Review Batch Test Summary

After running the batch test, we typically get a summary like this:

Example Console Output (Fig 1)

Fig 1

```
2025-02-10 12:00:00 - INFO - Test Case 1: ✓Pass | Input: Hello, World! | Output: hello world
2025-02-10 12:00:00 - INFO - Test Case 2: ✓Pass | Input: This is a TEST. | Output: this is a test
2025-02-10 12:00:00 - INFO - Test Case 3: ✓Pass | Input: 123 ABC! | Output: 123 abc
2025-02-10 12:00:00 - ERROR - Test Case 4: ✗Fail | Input: Python 3.9, is great! | Expected: python 39
is great | Got: python 39is great
2025-02-10 12:00:00 - INFO - Test Case 5: ✓ Pass | Input: $1000 in cash | Output: 1000 in cash

2025-02-10 12:00:00 - INFO - Batch Test Summary: 4 Passed, 1 Failed
```

Step 2: Identify Failed Test Cases

- From the summary, we see 1 failed test case (Test Case 4).
- Common failure reasons:
  - Incorrect formatting
  - Missing characters
  - Extra spaces or symbols

Extracting failed test cases for deeper analysis (Fig 2)

Fig 2

```
failed_cases = [
    ("Python 3.9, is great!", "python 39 is great", "python 39is great")
]

for input_text, expected_output, actual_output in failed_cases:
    print(f"\n🔴 **Failed Test Case Analysis**")
    print(f"Input: {input_text}")
    print(f"Expected Output: {expected_output}")
    print(f"Actual Output: {actual_output}")
```

© NIMI NOT TO BE REPUBLISHED

Expected Console Output (Fig 3)

Fig 3

```
🔴 **Failed Test Case Analysis**
Input: Python 3.9, is great!
Expected Output: python 39 is great
Actual Output: python 39is great
```

Issue Identified: The space after "39" is missing in the actual output.

Step 3: Categorize Failures

Failures can be grouped into different categories. (Fig 4)

Fig 4

Failure Type	Example Issue	Solution
Formatting	Missing spaces (e.g., `python 39is great`)	Adjust text processing logic
Incorrect Removal	Removed necessary characters	Modify regex cleanup
Unexpected Characters	Extra/missing symbols	Debug post-processing step

Step 4: Debugging the Root Cause

Check Function Logic (Python Debugging Example)

Let's modify the function to fix the spacing issue. (Fig 5)

Fig 5

```python
import re

def text_processing_fixed(input_text):
    """Fix spacing issue when removing punctuation."""
    return re.sub(r'[^\w\s]', '', input_text).lower().strip()

# Test the fixed function
print(text_processing_fixed("Python 3.9, is great!"))
```

© NIMI NOT TO BE REPUBLISHED

Expected Fixed Output (Fig 6)

Fig 6

```
python 39 is great
```

Step 5: Re-run the Batch Test After Fixes

After modifying the function, re-run the batch test to check if the issue is resolved. (Fig 7)

Fig 7

```
batch_test()
```

Expected Console Output (All Tests Passed) (Fig 8)

Fig 8

```
2025-02-10 12:30:00 - INFO - Batch Test Summary: 5 Passed, 0 Failed
```

Step 6: Store and Visualize Results

(1) Save Results to a CSV File

To track results for future analysis. (Fig 9)

Fig 9

```
import csv

results = [
    ["Test Case", "Input", "Expected Output", "Actual Output", "Status"],
    [1, "Hello, World!", "hello world", "hello world", "Pass"],
    [2, "This is a TEST.", "this is a test", "this is a test", "Pass"],
    [3, "123 ABC!", "123 abc", "123 abc", "Pass"],
    [4, "Python 3.9, is great!", "python 39 is great", "python 39is great", "Fail"],
    [5, "$1000 in cash", "1000 in cash", "1000 in cash", "Pass"]
]

with open("batch_test_results.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(results)

print("Results saved to batch_test_results.csv")
```

(2) Visualize Results with a Bar Chart

We can use **matplotlib** to show test results. (Fig 10)

Fig 10

```
import matplotlib.pyplot as plt

labels = ["Passed", "Failed"]
counts = [4, 1]

plt.bar(labels, counts, color=['green', 'red'])
plt.xlabel("Test Results")
plt.ylabel("Number of Tests")
plt.title("Batch Test Results")
plt.show()
```

Expected Output: A bar chart showing 4 Passed and 1 Failed.

Final Step: Generate a Report

You can generate a report in Markdown, a PDF, or an HTML file:

(1) Generate an HTML Report Using Python (Fig 11)

Fig 11

```
html_report = f"""
<html>
<head><title>Batch Test Report</title></head>
<body>
    <h1>Batch Test Summary</h1>
    <p><b>Total Tests:</b> 5</p>
    <p><b>Passed:</b> 4</p>
    <p><b>Failed:</b> 1</p>
</body>
</html>
"""

with open("batch_test_report.html", "w") as file:
    file.write(html_report)

print("Report saved as batch_test_report.html")
```

Final Report Output

- **CSV File:** batch_test_results.csv
- **HTML Report:** batch_test_report.html
- **Graph:** A visual representation of results.

© NIMI NOT TO BE REPUBLISHED

Batch test tips

Objective: At the end of this exercise you shall be able to
- **optimize batch tests for efficiency and reliability.**

Requirements

Tool/Instruments

- A working PC - as reqd.

PROCEDURE

Batch Test Tips: Step-by-Step Guide with Output

Batch testing is crucial for automating the evaluation of multiple test cases efficiently. Below are step-by-step **best practices, tips,** and **example outputs** to improve your batch testing process.

Step 1: Define Clear Test Cases

Before running a batch test, ensure you have well-defined test cases with clear input-output expectations.

Example Test Cases (Fig 1)

Fig 1

Test Case	Input	Expected Output
1	`"Hello, World!"`	`"hello world"`
2	`"Python 3.9 is great!"`	`"python 39 is great"`
3	`"$1000 in cash!"`	`"1000 in cash"`

Tip: Store test cases in a **CSV file or database** for easier updates and maintenance.

Step 2: Implement Logging for Each Test Case

Instead of manually tracking results, use logging to store outputs.

© NIMI NOT TO BE REPUBLISHED

Python Example (Fig 2)

Fig 2

```python
import logging

# Configure logging
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")

def text_processing(input_text):
    """Function to preprocess text: lowercase and remove punctuation."""
    import re
    return re.sub(r'[^\w\s]', '', input_text).lower()

# Sample test cases
test_cases = [
    ("Hello, World!", "hello world"),
    ("Python 3.9 is great!", "python 39 is great"),
    ("$1000 in cash!", "1000 in cash"),
]

def batch_test():
    passed, failed = 0, 0

    for i, (input_text, expected_output) in enumerate(test_cases, start=1):
        result = text_processing(input_text)
        if result == expected_output:
            logging.info(f"Test Case {i}: ✅ Pass | Input: {input_text} | Output: {result}")
            passed += 1
        else:
            logging.error(f"Test Case {i}: ❌ Fail | Input: {input_text} | Expected: {expected_output} | Got: {result}")
            failed += 1

    logging.info(f"\nBatch Test Summary: {passed} Passed, {failed} Failed")

# Run batch test
batch_test()
```

Tip: Log results to a **file** instead of just printing. (Fig 3)

Fig 3

```python
logging.basicConfig(filename="batch_test_results.log", level=logging.INFO)
```

Step 3: Capture and Analyze Batch Test Output

Example Console Output (Fig 4)

Fig 4

2025-02-10 12:00:00 - INFO - Test Case 1: ✓Pass | Input: Hello, World! | Output: hello world
2025-02-10 12:00:00 - INFO - Test Case 2: ✓Pass | Input: Python 3.9 is great! | Output: python 39 is great
2025-02-10 12:00:00 - INFO - Test Case 3: ✗Fail | Input: $1000 in cash! | Expected: 1000 in cash | Got: 1000in cash

2025-02-10 12:00:00 - INFO - Batch Test Summary: 2 Passed, 1 Failed

Tip: If a test fails, store expected vs. actual results in a separate error log file.

Step 4: Debug Failed Cases

When a test fails, analyze the failure reasons. (Fig 5)

Fig 5

Test Case	Expected Output	Actual Output	Issue
3	`"1000 in cash"`	`"1000in cash"`	Missing space after 1000

Tip: If failures occur due to formatting issues, **use debugging print statements.** (Fig 6)

Fig 6

```python
print(f"Before Processing: {input_text}")
print(f"After Processing: {result}")
```

© NIMI NOT TO BE REPUBLISHED

Step 5: Optimize Performance for Large Batches

If running tests on large datasets, optimize performance:

Use multiprocessing for parallel execution (Fig 7)

Fig 7

```python
import multiprocessing

def run_test_case(test_case):
    input_text, expected_output = test_case
    result = text_processing(input_text)
    return result == expected_output

# Run tests in parallel
with multiprocessing.Pool() as pool:
    results = pool.map(run_test_case, test_cases)

print(f"Total Passed: {results.count(True)}, Total Failed: {results.count(False)}")
```

Tip: For thousands of tests, store results in a database (SQL/NoSQL) instead of logs.

Step 6: Visualize Batch Test Results

Instead of only logs, use charts to summarize results.

Python Code to Generate a Bar Chart (Fig 8)

Fig 8

```python
import matplotlib.pyplot as plt

labels = ["Passed", "Failed"]
counts = [2, 1]

plt.bar(labels, counts, color=['green', 'red'])
plt.xlabel("Test Results")
plt.ylabel("Number of Tests")
plt.title("Batch Test Results")
plt.show()
```

Tip: Save the chart as an image for reporting (Fig 9)

Fig 9

```python
plt.savefig("batch_test_chart.png")
```

Step 7: Automate and Schedule Batch Tests

- **For Continuous Integration (CI):** Use **Jenkins, GitHub Actions, or CI/CD pipelines**
- **For Scheduled Testing:** Use **cron jobs** in Linux (Fig 10)

Fig 10

```
0 0 * * * python batch_test.py  # Runs daily at midnight
```

Tip: If running batch tests on cloud servers, use **AWS Lambda or Google Cloud Functions** for automation.

© NIMI NOT TO BE REPUBLISHED

Software Testing Assistant - Automating Test Execution

What happens when the user interface changes?

Objective: At the end of this exercise you shall be able to
- **user interface changes improve usability and experience.**

Requirements	
Tool/Instruments	
• A working PC with UI Design Tools	- as reqd.

PROCEDURE

When a user interface (UI) changes, it typically follows a structured process, whether in web, mobile, or desktop applications. Below is a step-by-step breakdown of what happens, along with expected outputs at each stage:

1 User Interaction (Trigger Event)
- The user interacts with the UI (e.g., clicks a button, hovers over an element, types text).
- **Example Output**
  - Mouse click event detected on "Submit" button.
  - Keyboard event: "A" key pressed in input field.

2 Event Handling
- The UI framework or JavaScript (in web apps) captures the event.
- Event listeners call the appropriate function.
- **Example Output**
  - onClick() function triggered for "Submit" button.
  - onChange() detected text input modification.

3 State Update (If Applicable)
- If the UI is state-driven (e.g., React, Vue, or Flutter), the event updates the application state.
- The state manager (e.g., Redux, Context API, Vuex) modifies the stored data.
- **Example Output**
  - setState({ isLoading: true }) updates loading state.
  - store.dispatch({ type: "UPDATE_USERNAME", payload: "JohnDoe" }).

4 Virtual DOM Reconciliation (For React-like Frameworks)
- The new state triggers a re-render of the components.
- The virtual DOM compares the new UI with the previous one (diffing).
- Only changed parts of the UI update efficiently.
- **Example Output**
  - React diffing detected changes in div#profile-name.
  - Virtual DOM updated component <UserProfile>.

5 UI Re-render & DOM Update
- The UI updates to reflect the new state.
- The browser modifies the actual DOM (or a UI framework updates its equivalent).
- **Example Output**
  - The text inside <h1> changes from "Welcome, Guest" to "Welcome, JohnDoe".
  - A new button appears dynamically: <button>Logout</button>.

6 Style and Animation Changes
- CSS changes may be applied due to class modifications.
- Transitions, animations, and effects occur.
- **Example Output**
  - Button color changes to blue (.btn-primary applied).
  - A modal window smoothly fades in using opacity: 1 animation.

7 Backend API Calls (If Required)
- If the UI change involves fetching or sending data, an API request is triggered.
- The response updates the state again.
- **Example Output:**
  - Fetching user profile: GET /api/user/123.
  - Server response: { "name": "John Doe", "age": 30 }.

8 UI Finalization
- The new UI is fully displayed.
- Any loaders, spinners, or placeholders are removed.
- **Example Output**
  - Loading spinner hidden.
  - Success message: "Profile updated successfully!".

9 Accessibility & Responsiveness Adjustments
- Adjustments happen for different screen sizes or assistive technologies.
- **Example Output**
  - Mobile layout switches to single-column.
  - Screen reader announces: "Button clicked, form submitted".

10 User Feedback and Additional Interactions
- The user sees the updated interface and may take further actions.
- **Example Output**
  - User clicks another button, repeating the cycle.

© NIMI NOT TO BE REPUBLISHED

Editing object descriptions in the GUI map adding GUI objects to the GUI map

Objective: At the end of this exercise you shall be able to
- **editing object descriptions and adding objects enhance GUI functionality.**

Requirements

Tool/Instruments
- A working PC GUI Design and Code Editors - as reqd.

PROCEDURE

Editing Object Descriptions & Adding GUI Objects to the GUI Map

When working with a GUI Map (often used in test automation tools like UFT, Selenium, or custom UI mapping systems), objects are stored and identified by their properties. Here's a step-by-step guide for **editing object descriptions and adding new GUI objects** to the GUI map.

Part 1: Editing Object Descriptions in the GUI Map

When modifying object descriptions (e.g., changing identifiers like name, XPath, or CSS selector), the following steps occur.

1 **Open the GUI Map or Object Repository**
- Load the existing GUI Map in a test automation tool.
- **Example Output**
  - "GUI Map loaded successfully."
  - List of existing objects displayed.

2 **Select the Object to Edit**
- Identify the object whose description needs to be changed.
- Select it from the list or hierarchy view.
- **Example Output**
  - "Button_Login selected."

3 **Edit Object Properties**
- Modify attributes such as name, XPath, CSS selector, or index.
- **Example Output**
  - Before: Button_Login { xpath: //button[text()='Login'] }
  - After: Button_Login { xpath: //button[@id='login-btn'] }
  - "Object description updated."

4 **Save Changes**
- Apply and save modifications to the GUI Map.
- **Example Output**
  - "GUI Map saved successfully."
  - "Button_Login updated with new identifier."

5 **Test the Updated Object**
- Run an automated test or manually verify the updated object.
- **Example Output**
  - "Test executed: Login button found successfully."

Part 2: Adding GUI Objects to the GUI Map

When adding new GUI objects, the following steps are followed.

1 **Open GUI Map Editor**
- Launch the GUI Map/Object Repository tool.
- **Example Output**
  - "GUI Map Editor opened."

2 **Start Object Identification**
- Use Object Spy or Recording Mode to capture a new object.
- **Example Output**
  - "Object Spy active. Hover over the object to capture."

3 **Capture Object Properties**
- Click on the UI element (button, textbox, dropdown) to capture details.
- **Example Output**
  - Captured properties for "Search_Box":
    - Type: input
    - ID: search-box
    - XPath: //input[@id='search-box']

4 **Add the Object to the GUI Map**
- Assign a unique name and store the object in the repository.
- **Example Output**
  - "New object added: Search_Box."
  - "Stored properties: { id: 'search-box', type: 'input' }"

5 **Verify Object in the Map**
- Open the GUI Map and confirm the object is listed.
- **Example Output**
  - "Object 'Search_Box' is now visible in the repository."

6 **Save the GUI Map**
- Save the modified GUI Map with new objects.
- **Example Output**
  - "GUI Map updated successfully."

7 **Validate the New Object in a Test**
- Run a test script to check if the object is correctly recognized.
- **Example Output**
  - Test script: FindElement(Search_Box). sendKeys("AI Tools")
  - "Test Passed: Search box identified and input entered."

© NIMI NOT TO BE REPUBLISHED

Updating the GUI map with the run wizard

Objective: At the end of this exercise you shall be able to
- **updating the GUI map streamlines user interface configuration.**

Requirements

Tool/Instruments

- A working PC Run Wizard Tool - as reqd.

PROCEDURE

Updating the GUI Map Using the Run Wizard – Step-by-Step Guide

In **test automation tools like UFT (Unified Functional Testing),** the **Run Wizard** helps update the **GUI Map (Object Repository)** by detecting changes in UI objects and updating their properties. Below is a structured process for updating the GUI Map using the Run Wizard.

Step 1: Launch the Test Automation Tool

- Open **UFT (Unified Functional Testing)** or any relevant test automation tool.
- **Example Output**
  - "UFT loaded successfully."
  - "GUI Map available for test project."

Step 2: Open the Object Repository

- Navigate to Resources → Object Repository.
- **Example Output**
  - "Object Repository opened."
  - "Existing GUI objects displayed."

Step 3: Start the Run Wizard

- Select Tools → Update Run Wizard (in UFT).
- The Run Wizard launches and begins analyzing the test script and GUI Map.
- **Example Output**
  - "Update Run Wizard started."
  - "Scanning test objects and application under test."

Step 4: Select the Test Case for GUI Update

- Choose a test script that interacts with the application's GUI.
- **Example Output**
  - "Test case 'LoginTest' selected."
  - "Objects in 'LoginTest' are being verified."

Step 5: Execute the Run Wizard to Identify UI Changes

- The wizard runs the test and compares the existing GUI Map with the current UI.
- It detects missing, changed, or new objects.
- **Example Output**
  - "Scanning UI for changes..."
  - "Object 'LoginButton' properties mismatch detected."
  - "Suggested update: XPath changed from //button[@name='login'] to //button[@id='login-btn']."

Step 6: Review and Confirm Changes

- The Run Wizard presents a list of modified and new objects.
- Users review and accept/reject updates.
- **Example Output**
  - "Changes detected: 3 objects modified, 1 new object found."
  - "LoginButton → XPath Updated."
  - "SearchBox → New object added."

Step 7: Apply Changes to the GUI Map

- Confirm the updates to store changes in the Object Repository.
- **Example Output**
  - "Applying changes..."
  - "GUI Map updated successfully."

Step 8: Save and Close

- Save the updated GUI Map and close the Run Wizard.
- **Example Output**
  - "Object Repository saved."
  - "Run Wizard completed successfully."

Step 9: Validate the Updated GUI Map

- Run an automated test to verify if the newly updated GUI objects work correctly.
- **Example Output**
  - Test script: Click(LoginButton);
  - "Test Passed: LoginButton clicked successfully."

© NIMI NOT TO BE REPUBLISHED

Load test planning

Objective: At the end of this exercise you shall be able to
- **ensure system performance under expected load conditions.**

Requirements

Tools/Equipment/Instruments

- A working PC with Apache JMeter - as reqd.

PROCEDURE

Load Test Planning – Step-by-Step Guide with Output

Load testing is crucial for evaluating an application's performance under expected user load conditions. Here's a step-by-step guide on how to plan and execute a load test, along with the expected outputs.

Step 1: Define Load Test Objectives

Before executing a load test, establish its goals based on business and technical requirements.

- Identify performance benchmarks (e.g., response time, throughput, error rate).
- Define maximum concurrent users expected.
- Determine critical transactions (e.g., login, checkout, API calls).

Example Output

- "Load test objective: Ensure system supports 500 concurrent users with response time < 2 sec."
- "Critical transactions: User login, product search, checkout."

Step 2: Identify Performance Metrics

Select key performance indicators (KPIs) to monitor during the load test.

Common Metrics

- Response Time (average, peak)
- Throughput (requests per second)
- CPU & Memory Usage
- Error Rate (%)
- Database Query Execution Time

Example Output

- "Selected KPIs: Response time, CPU usage, error rate, and database latency."
- "Response time target: < 2 seconds for 95% of users."

Step 3: Define Load Test Scenarios

A test scenario simulates real-world user actions.

- Identify user journeys (e.g., login, search, checkout).
- Define test data (e.g., usernames, product IDs).
- Determine load distribution (e.g., 70% search, 30% checkout).

Example Output

- "Scenario 1: 70% users searching products, 30% users purchasing."
- "Scenario 2: 100% concurrent users logging in at once."

Step 4: Select Load Testing Tools

Choose an appropriate load testing tool based on the tech stack.

Popular Load Testing Tools

- J Meter (Open-source, UI & API testing)
- Load Runner (Enterprise-grade performance testing)
- Gatling (Scala-based, for high-performance systems)
- Locust (Python-based, simple scripting)

Example Output

- "Selected Tool: JMeter for HTTP-based load testing."
- "Alternative: Locust for distributed load testing."

Step 5: Configure the Load Test Environment

Set up the infrastructure to run the test.

- Choose test environments (staging/production).
- Ensure test data availability.
- Configure load generator servers.

Example Output

- "Load testing will run on staging servers with a replica of production data."
- "Allocated 3 cloud-based load generators for traffic simulation."

© NIMI NOT TO BE REPUBLISHED

Step 6: Define Load Testing Strategy

Decide how to apply the load.

Types of Load Tests

- Baseline Test – Light load to measure basic performance
- Stress Test – Push system beyond limits
- Spike Test – Sudden increase in users
- Soak Test – Long-duration load test

Example Output

- "Load Strategy: Ramp-up from 50 to 500 users over 10 minutes."
- "Spike test: Instant jump from 100 to 1000 users in 30 seconds."

Step 7: Execute the Load Test

Run the test based on the defined scenarios.

- Start load generators.
- Monitor system resources (CPU, memory, database).
- Log failures & timeouts.

Example Output

- "Test execution started at 10:00 AM."
- "Current load: 300 concurrent users, CPU at 60% usage."
- "Detected 5% timeout errors on checkout page."

Step 8: Analyze Load Test Results

Evaluate test logs and reports to identify bottlenecks.

- Check response time trends.
- Identify failed transactions.
- Analyze CPU, memory, and database performance.

Example Output

- "Peak response time: 3.2 seconds (above target)."
- "Database queries taking longer than expected (avg. 800ms)."
- "Optimization needed for API response under heavy load."

Step 9: Optimize & Retest

Based on the results, optimize and re-run the test.

Optimization Areas

- Increase server resources (CPU, RAM, DB scaling).
- Optimize database queries.
- Implement caching (e.g., Redis, CDN).

Example Output

- "Increased server capacity: 4 CPUs → 8 CPUs."
- "Enabled caching for repeated queries."
- "Re-running load test with updated settings."

Step 10: Document & Report Findings

Summarize findings in a load test report.

Key Report Sections

- Test Objectives & Scenarios
- Performance Metrics (Before & After)
- System Bottlenecks Identified
- Recommendations & Next Steps

Example Output

- "Load Test Report generated."
- "Summary: Performance improved after DB optimization, but checkout latency still needs improvement."

© NIMI NOT TO BE REPUBLISHED

The LoadRunner controller at a glance

Objective: At the end of this exercise you shall be able to
- **manage, execute, and monitor load testing scenarios.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

LoadRunner Controller at a Glance – Step-by-Step Guide with Output

The LoadRunner Controller is a critical component in LoadRunner (by Micro Focus) that helps design, execute, and monitor performance tests in real-time. It controls multiple virtual users (Vusers) to simulate load and measures system behavior under stress. Below is a step-by-step guide with expected outputs.

Step 1: Launch LoadRunner Controller

- Open LoadRunner Professional or LoadRunner Enterprise.
- Select Controller from the LoadRunner tools.

Example Output

- "LoadRunner Controller started successfully."
- "Welcome screen displayed with options to create or load a scenario."

Step 2: Create a New Test Scenario

- Choose New Scenario → Select Manual Scenario or Goal-Oriented Scenario.
- Manual Scenario → Define Vuser groups and load settings manually.
- Goal-Oriented Scenario → Set performance goals (e.g., response time, throughput).

Example Output

- "New Scenario created: 'E-commerce Load Test'."
- "Scenario Type: Goal-Oriented (Target response time < 2s)."

Step 3: Add Load Test Scripts

- Click Add Script → Select a script recorded in VuGen (Virtual User Generator).
- Choose scripts for different user actions (e.g., Login, Search, Checkout).

Example Output

- "Script 'Login_Scenario' added with 200 Vusers."
- "Script 'Checkout_Process' added with 100 Vusers."

Step 4: Configure Load Generators

- Assign Load Generators (LGs) to distribute test traffic.
- Choose on-premise or cloud-based LGs.

Example Output

- "3 Load Generators added: LG1 (USA), LG2 (Europe), LG3 (Asia)."
- "Connection status: All Load Generators available."

Step 5: Define Load Test Execution Settings

- Set ramp-up policy (how users are gradually added).
- Define test duration (steady-state run time).
- Set ramp-down policy (how users exit the test).

Example Output

- "Ramp-up: Increase from 50 to 500 Vusers over 10 minutes."
- "Steady-state duration: 30 minutes."
- "Ramp-down: Decrease users over 5 minutes."

Step 6: Configure Runtime Settings

- Define think time (simulated user delay between actions).
- Enable correlation for dynamic values (e.g., session IDs).
- Set network bandwidth limitations (e.g., 3G, 4G, Wi-Fi).

Example Output

- "Think time enabled: Random delay 2-5 seconds."
- "Network simulation: Users split (50% Wi-Fi, 30% 4G, 20% 3G)."

Step 7: Start Load Test Execution

- Click Start Scenario to begin the load test.
- Monitor real-time performance metrics.

© NIMI NOT TO BE REPUBLISHED

Example Output

- "Load test started with 500 Vusers."
- "CPU Usage: 40% | Memory Usage: 3GB | Transactions/sec: 150."

Step 8: Monitor Test Execution in Real-Time

- Use LoadRunner Dashboard to track KPIs.
- Key Metrics
  - Transaction Response Time
  - Throughput (requests/sec)
  - Error Rate (%)
  - Server Resource Utilization (CPU, Memory, Disk, Network)

Example Output

- "Current Load: 350 Vusers."
- "Average Response Time: 1.8s (within limits)."
- "Error Rate: 2% (acceptable range)."

Step 9: Stop the Load Test

After execution, stop the scenario and generate test reports.

Example Output

- "Load test completed successfully."
- "Final statistics: Peak Load = 500 Vusers, Max Response Time = 2.3s."

Step 10: Analyze Load Test Results

- Open Analysis Tool to generate performance reports.
- Identify bottlenecks and slow transactions.
- Compare actual vs. expected performance.

Example Output

- "Report generated: Average Response Time = 1.9s."
- "Checkout page slowest at peak load (3.5s). Optimization needed."

Step 11: Optimize System & Retest

- Apply fixes (e.g., database indexing, caching, load balancing).
- Re-run the test to verify improvements.

Example Output

- "Optimization: Database queries reduced from 500ms → 200ms."
- "Re-running test with updated configurations."

© NIMI NOT TO BE REPUBLISHED

Creating a scenario

Objective: At the end of this exercise you shall be able to
- **define and configure virtual users for testing.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Creating a Scenario in LoadRunner – Step-by-Step Guide with Output

A scenario in LoadRunner Controller defines the conditions for executing a load test, such as the number of virtual users (Vusers), test scripts, load distribution, and performance metrics. Below is a step-by-step guide to creating a scenario with expected outputs.

Step 1: Launch LoadRunner Controller

- Open LoadRunner Professional or LoadRunner Enterprise.
- Select Controller from the LoadRunner tools.

Example Output

- "LoadRunner Controller started successfully."
- "Welcome screen displayed with options to create or load a scenario."

Step 2: Create a New Scenario

- Click File → New Scenario.
- Select Scenario Type:
  - Manual Scenario – Manually configure Vusers and load distribution.
  - Goal-Oriented Scenario – Define a performance goal (e.g., response time < 2s).

Example Output

- "New Scenario Wizard opened."
- "Scenario Type: Goal-Oriented Scenario selected (Target response time < 2s)."

Step 3: Add Virtual User Scripts (Vuser Scripts)

- Click Add Script → Select scripts recorded in VuGen (Virtual User Generator).
- Assign scripts for different transactions (e.g., login, search, checkout).

Example Output

- "Script 'Login_Scenario' added with 200 Vusers."
- "Script 'Checkout_Process' added with 100 Vusers."

Step 4: Configure Load Generators

- Assign Load Generators (LGs) to distribute the test load.
- Choose between on-premise or cloud-based LGs.

Example Output

- "3 Load Generators added: LG1 (USA), LG2 (Europe), LG3 (Asia)."
- "Connection status: All Load Generators available."

Step 5: Define Load Distribution & Vuser Groups

- Set the number of Vusers per script.
- Define how users are distributed across different Load Generators.

Example Output

- "Vuser Distribution:
- 200 Vusers → Login Scenario
- 100 Vusers → Checkout Process
- 50 Vusers → Product Search"

Step 6: Configure Load Test Execution Settings

- **Ramp-up Policy:** Gradual increase in Vusers over time.
- **Test Duration:** Define steady-state execution time.
- **Ramp-down Policy:** Gradual reduction of users.

Example Output

- **"Ramp-up:** Increase from 50 to 500 Vusers over 10 minutes."
- **"Steady-state duration:** 30 minutes."
- **"Ramp-down:** Decrease users over 5 minutes."

Step 7: Configure Runtime Settings

- Set think time (simulated user delay between actions).
- Enable correlation for dynamic values (e.g., session IDs).

© NIMI NOT TO BE REPUBLISHED

- Set network bandwidth limitations (e.g., 3G, 4G, Wi-Fi).

Example Output

- "Think time enabled: Random delay 2-5 seconds."
- "Network simulation: Users split (50% Wi-Fi, 30% 4G, 20% 3G)."

Step 8: Save and Name the Scenario

- Click File → Save As.
- Name the scenario (e.g., Ecommerce_LoadTest.lrs).

Example Output

- "Scenario saved as Ecommerce_LoadTest.lrs."

Step 9: Verify Scenario Configuration

- Review all scenario settings before execution.
- Ensure that load generators are active and scripts are correctly assigned.

Example Output

- "Scenario verification passed: All scripts and Load Generators ready."

Step 10: Execute the Scenario

- Click Start Scenario to begin the test.
- Monitor real-time performance metrics.

Example Output

- "Load test started with 500 Vusers."
- "CPU Usage: 40% | Memory Usage: 3GB | Transactions/sec: 150."

Step 11: Monitor & Analyze Performance Metrics

- Use LoadRunner's real-time dashboard to track key metrics:
  - Transaction Response Time
  - Throughput (requests/sec)
  - Error Rate (%)
  - Server Resource Utilization (CPU, Memory, Network)

Example Output

- "Current Load: 350 Vusers."
- "Average Response Time: 1.8s (within limits)."
- "Error Rate: 2% (acceptable range)."

Step 12: Stop the Scenario & Save Results

- Once the test duration completes, stop the scenario.
- Save test logs and generate performance reports.

Example Output

- "Load test completed successfully."
- "Final statistics: Peak Load = 500 Vusers, Max Response Time = 2.3s."

Step 13: Analyze Load Test Results

- Open LoadRunner Analysis Tool to generate detailed reports.
- Identify bottlenecks and slow transactions.

Example Output

- "Report generated: Average Response Time = 1.9s."
- "Checkout page slowest at peak load (3.5s). Optimization needed."

Step 14: Optimize System & Retest

- Apply fixes (e.g., database indexing, caching, load balancing).
- Re-run the scenario to verify improvements.

Example Output

- "Optimization: Database queries reduced from 500ms → 200ms."
- "Re-running test with updated settings."

© NIMI NOT TO BE REPUBLISHED

Using rendezvous points

Objective: At the end of this exercise you shall be able to
- **synchronize virtual users to simulate real-time actions.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Using Rendezvous Points in LoadRunner – Step-by-Step Guide with Output

Rendezvous Points in LoadRunner help simulate peak user loads by instructing multiple virtual users (Vusers) to perform a specific action at the same time. This is useful for stress testing critical transactions, such as logging in, checking out, or booking a ticket.

Step 1: Open LoadRunner VuGen

- Launch LoadRunner VuGen (Virtual User Generator).
- Open or record a new script for the test scenario.

Example Output

- "VuGen started successfully."
- "Script 'Ecommerce_Checkout' loaded."

Step 2: Identify the Critical Transaction

- Select a transaction where you want all users to execute simultaneously.
- Example: "Checkout", where many users try to complete purchases at the same time.

Example Output

- "Identified critical transaction: User clicks 'Place Order' button."

Step 3: Insert a Rendezvous Point

- In VuGen, place the rendezvous point before the critical transaction.
- Use the lr_rendezvous() function to specify the point.

Steps in VuGen

1 Open the script editor.

2 Locate the section where you want the Vusers to synchronize.

3 Insert (Fig 1)

Fig 1

```
lr_rendezvous("Checkout_Rendezvous");
```

4 Save the script.

Example Output

- "Rendezvous point 'Checkout_Rendezvous' added before 'Place Order' action."

Step 4: Configure Rendezvous Policy in LoadRunner Controller

- Open LoadRunner Controller.
- Load the scenario containing the VuGen script.
- Click Design → Rendezvous to open the Rendezvous Editor.
- Add the rendezvous name (Checkout_Rendezvous).
- Set policy (e.g., how many users should be synchronized).

Example Policy Options

- Release all Vusers at once (ideal for stress tests).
- Release in batches (e.g., 50 users every 5 seconds).

Example Output

- "Rendezvous point 'Checkout_Rendezvous' added in LoadRunner Controller."
- "Configured policy: Release 100 Vusers simultaneously."

Step 5: Assign Rendezvous Points to Vuser Groups

- In Scenario Groups, select the Vuser Group.
- Click Assign Rendezvous and select 'Checkout_ Rendezvous'.
- Choose the number of Vusers that should participate.

Example Output

- "Rendezvous 'Checkout_Rendezvous' assigned to 200 Vusers."

Step 6: Execute the Load Test with Rendezvous Point

- Start the test scenario in LoadRunner Controller.
- Monitor the Rendezvous status in real-time.

© NIMI NOT TO BE REPUBLISHED

Example Output

- "Scenario started with 500 Vusers."
- "200 Vusers waiting at 'Checkout_Rendezvous' before proceeding."

Step 7: Monitor Rendezvous Execution in Real-Time

- Go to the Rendezvous Graph in the LoadRunner Controller.
- Check the status of users waiting at the rendezvous point.
- Ensure all users execute the critical transaction simultaneously.

Example Output

- "Rendezvous Status: 200 Vusers waiting, 100 released so far."
- "Peak transaction load achieved: 200 checkout requests in 1 second!"

Step 8: Analyze Load Test Results

- Open LoadRunner Analysis Tool after the test.
- Check the effect of peak loads on response time and server performance.

Example Output

- "Checkout response time increased from 1.5s to 3.8s under peak load."
- "CPU usage spiked to 90% during the rendezvous event."

Step 9: Optimize and Retest

- Apply fixes (e.g., increase server capacity, optimize database queries).
- Re-run the load test with the rendezvous point.

Example Output

- "Optimization: Checkout query execution time reduced from 800ms → 300ms."
- "Re-running load test to validate improvements."

© NIMI
NOT TO BE REPUBLISHED

Configuring a scenario

Objective: At the end of this exercise you shall be able to
- **set up test parameters and virtual user behaviour.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Configuring a Scenario in LoadRunner – Step-by-Step Guide with Output

After creating a scenario in LoadRunner Controller, the next step is to configure it by adjusting Vuser groups, runtime settings, load generators, and test execution policies. This ensures that the performance test runs as per the desired load conditions.

Step 1: Open LoadRunner Controller

- Launch LoadRunner Controller from the Micro Focus LoadRunner toolset.
- Open an existing scenario or create a new scenario.

Example Output

- "LoadRunner Controller started successfully."
- "Scenario 'Ecommerce_LoadTest.lrs' loaded."

Step 2: Add Load Test Scripts

- Click Add Script to add pre-recorded scripts from VuGen.
- Assign different scripts for multiple transactions (e.g., Login, Search, Checkout).

Example Output

- "Script 'Login_Scenario' added with 200 Vusers."
- "Script 'Checkout_Process' added with 100 Vusers."

Step 3: Configure Load Generators

- Navigate to Load Generators in the Controller.
- Assign one or more Load Generators to distribute test traffic.
- Click Connect to validate the Load Generator status.

Example Output

- "Load Generators added: LG1 (USA), LG2 (Europe)."
- "Connection status: All Load Generators available."

Step 4: Configure Vuser Groups

- Click Scenario Groups → Configure Vuser Distribution.
- Assign number of Vusers per script.

Example Output

- "Vuser Distribution
  - 200 Vusers → Login Scenario
  - 100 Vusers → Checkout Process
  - 50 Vusers → Product Search"

Step 5: Set Load Distribution

- Choose Manual Load Test or Goal-Oriented Scenario.
- Define the number of Vusers per load generator.

Example Output

"Manual Load Test selected: 500 total Vusers distributed across 2 Load Generators."

Step 6: Configure Load Test Execution Policy Open Schedule Editor → Configure test execution settings

- **Ramp-up:** How quickly users are added.
- **Steady-state duration:** Time the load is maintained.
- **Ramp-down:** Gradual reduction of users.

Example Policy Configuration

- **Ramp-up:** 50 to 500 Vusers over 10 minutes.
- **Steady-state duration:** 30 minutes.
- **Ramp-down:** Decrease users over 5 minutes.

Example Output

- "**Ramp-up:** Increase from 50 to 500 Vusers over 10 minutes."
- "**Steady-state duration:** 30 minutes."
- "**Ramp-down:** Decrease users over 5 minutes."

Step 7: Configure Runtime Settings

- Click Run-Time Settings in the Controller.
- Adjust the following settings
  - Think Time (User delay between actions).
  - Pacing (Time between iterations).

© NIMI NOT TO BE REPUBLISHED

- Error Handling (Continue on error, retry on failure).

Example Output

- "**Think time enabled:** Random delay between 2-5 seconds."
- "**Iteration pacing:** 10 seconds between transactions."
- "**Error Handling:** Continue test on minor errors."

Step 8: Configure Rendezvous Points (Optional)

- If needed, add rendezvous points to synchronize Vusers for peak load testing.

Example Output

- "Rendezvous 'Checkout_Rendezvous' added for 200 Vusers."

Step 9: Save the Scenario Configuration

- Click File → Save As and save the scenario.

Example Output

- "Scenario configuration saved as 'Ecommerce_LoadTest.lrs'."

Step 10: Verify Configuration and Start Load Test

- Click Verify Scenario to check for configuration errors.
- Click Start Scenario to begin the test.

Example Output

- "Scenario verification passed: All scripts and Load Generators ready."
- "Load test started with 500 Vusers."

© NIMI
NOT TO BE REPUBLISHED

Configuring a host

Objective: At the end of this exercise you shall be able to
- **assign and manage machines for load testing execution.**

Requirements	
Tools/Equipment/Instruments	
• A working PC with load runner controller	- as reqd.

PROCEDURE

Configuring a Host in LoadRunner Controller: Step-by-Step Guide

In LoadRunner Controller, a host (Load Generator) is a machine that runs virtual users (VUsers) during a performance test. Proper host configuration ensures accurate load distribution and efficient test execution.

Follow this step-by-step guide to configure a host (Load Generator) in LoadRunner Controller, including expected output at each stage.

Step 1: Launch LoadRunner Controller

Actions

- Open LoadRunner Controller from the Start menu or LoadRunner Launcher.
- Create a new scenario or load an existing one.

Output

- LoadRunner Controller opens, ready for host configuration.

Step 2: Open Load Generator Configuration

Actions

- Click "Load Generators" in the Controller window.
- The Load Generators dialog box appears, displaying available and assigned hosts.

Output:

- Load Generator window opens, showing host configuration options.

Step 3: Add a New Host (Load Generator)

Actions

- Click "Add" to configure a new host.
- Enter the Host Name or IP Address of the Load Generator machine.
- Click OK.

Output

- The host is added but not yet connected.

Step 4: Connect to the Host

Actions

- Select the newly added host from the list.
- Click "Connect" to establish a connection.
- If successful, the status changes to Ready.

Troubleshooting (if status remains Down)

- Ensure the LoadRunner Agent is running on the host.
- Check firewall settings to allow communication.
- Verify network connectivity between the Controller and Load Generator.

Output

- Host connection established, showing Ready status.

Step 5: Assign VUsers to the Host

Actions

- Click "Design" → "Scenario Groups".
- Assign the VUser groups to the configured host.
- Click OK to save changes.

Output

- Virtual Users (VUsers) are assigned to the host, ready for execution.

Step 6: Configure Host Settings

Actions

- Click "Edit Host" to modify settings.
- Adjust:
  - Maximum number of VUsers the host can handle.
  - Resource monitoring settings.
  - Load balancing options (if using multiple hosts).
  - Click OK to apply changes.

© NIMI NOT TO BE REPUBLISHED

Output

- Host settings are optimized for the load test.

Step 7: Validate Host Performance Before Execution

Actions

- Click "Run" → "Validate Hosts".
- LoadRunner performs a check to verify the host's capability.
- If validation is successful, the host is ready for execution.

If validation fails

- Check system CPU, memory, and network usage.
- Ensure the LoadRunner Agent Process is running.

Output

- Host is validated and ready for performance testing.

Step 8: Save Scenario with Configured Hosts

Actions

- Click "File" → "Save Scenario As".
- Enter a scenario name and save the file (.lrs).

Output

- Scenario with configured hosts is saved for future use.

Step 9: Execute the Scenario with Configured Hosts

Actions

- Click "Start Scenario" to begin execution.
- Monitor real-time graphs for
  - CPU & Memory Usage
  - Transaction Response Time
  - VUser Execution Status

Output

- Load test begins, utilizing configured hosts to generate load.

Step 10: Analyze Test Results

Actions

- Click "Stop Scenario" after completion.
- Save test logs and reports for further analysis.
- Use LoadRunner Analysis to examine:
- Response times under load.
- Server resource utilization.
- Errors and failures.

Output

- Test execution completes, and performance reports are generated.

© NIMI NOT TO BE REPUBLISHED

Preparing to run a scenario

Objective: At the end of this exercise you shall be able to
- **finalize configurations and ensure environment readiness for testing.**

Requirements
Tools/Equipment/Instruments
• A working PC with Load Runner Controller - as reqd.

PROCEDURE

Preparing to Run a Scenario in LoadRunner – Step-by-Step Guide with Output

Before running a scenario in LoadRunner Controller, you need to verify the setup, configure execution settings, and ensure all components are ready. This guide covers step-by-step preparations with expected outputs.

Step 1: Launch LoadRunner Controller

- Open LoadRunner Controller from the Micro Focus LoadRunner toolset.
- Load an existing scenario or create a new one.

Example Output

- "LoadRunner Controller started successfully."
- "Scenario 'Ecommerce_LoadTest.lrs' loaded."

Step 2: Verify Load Generators

- Navigate to Load Generators tab.
- Check the availability of all configured Load Generators (LGs).
- Click Connect to test connectivity.

Example Output

- "Load Generators detected: LG1 (USA), LG2 (Europe)."
- "Connection status: All Load Generators available."

Step 3: Verify Vuser Scripts and Runtime Settings

- Open the Vuser Scripts section to check the assigned scripts.
- Click Run-Time Settings to verify
  - Think Time (User delay between actions).
  - Pacing (Time between iterations).
  - Error Handling (Continue on error, retry on failure).

Example Output

- "Scripts verified: Login_Scenario, Checkout_Process, Product_Search."
- "Think time enabled: Random delay 2-5 seconds."
- "Iteration pacing: 10 seconds between transactions."

Step 4: Assign Load Distribution & Vuser Groups

Open Scenario Groups → Configure Vuser Distribution.

Assign the number of Vusers per script.

Example Output

- "Vuser Distribution:
  - 200 Vusers → Login Scenario
  - 100 Vusers → Checkout Process
  - 50 Vusers → Product Search"

Step 5: Configure Load Test Execution Policy

- Open Schedule Editor → Configure test execution settings:
- Ramp-up: How quickly users are added.
- Steady-state duration: Time the load is maintained.
- Ramp-down: Gradual reduction of users.

Example Configuration

- Ramp-up: 50 to 500 Vusers over 10 minutes.
- Steady-state duration: 30 minutes.
- Ramp-down: Decrease users over 5 minutes.

Example Output

- "Ramp-up: Increase from 50 to 500 Vusers over 10 minutes."
- "Steady-state duration: 30 minutes."
- "Ramp-down: Decrease users over 5 minutes."

Step 6: Configure Rendezvous Points (Optional)

- If needed, add rendezvous points to synchronize Vusers for peak load testing.

Example Output

- "Rendezvous 'Checkout_Rendezvous' added for 200 Vusers."

Step 7: Enable Real-Time Monitoring

© NIMI NOT TO BE REPUBLISHED

- Enable Performance Monitors for server, database, and network.
- Set up monitoring for CPU, memory, disk I/O, response time, and transaction rate.

Example Output

- "Monitoring enabled: CPU Usage, Memory Usage, Transaction Response Time."

Step 8: Verify Scenario Configuration

- Click Scenario → Verify Scenario to check for errors.
- Ensure all scripts, load generators, and settings are correctly configured.

Example Output

- "Scenario verification successful: No errors detected."
- "All Vusers assigned, scripts validated, load generators connected."

Step 9: Save the Scenario Configuration

- Click File → Save As and save the scenario.

Example Output

- "Scenario configuration saved as 'Ecommerce_ LoadTest.lrs'."

Step 10: Ready to Start the Load Test

- Click Start Scenario to begin the test.

Example Output

- "Scenario is now ready to execute."
- "Click 'Start Scenario' to begin load testing."

© NIMI
NOT TO BE REPUBLISHED

Managing scenarios using test director

Objective: At the end of this exercise you shall be able to
- **organize, execute, and track scenarios with test director.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Managing Scenarios Using TestDirector – Step-by-Step Guide with Output

TestDirector (now known as Micro Focus ALM/Quality Center) is used to manage test scenarios efficiently by storing, organizing, and executing LoadRunner test scripts. This guide outlines how to manage scenarios using TestDirector step by step with expected outputs.

Step 1: Launch TestDirector (ALM/Quality Center)

- Open TestDirector from your desktop or web browser.
- Enter your username and password, then click Login.
- Select the relevant Project where your test scenarios are stored.

Example Output

- "TestDirector (ALM) launched successfully."
- "Logged in as 'TestAdmin' to project 'Ecommerce_Testing'."

Step 2: Navigate to Test Plan Module

- In TestDirector, go to the Test Plan module.
- Browse or search for existing LoadRunner test scripts.
- Verify that all necessary test scripts are available and up to date.

Example Output

- "Test Plan module opened."
- "Scripts available: 'Login_Test', 'Checkout_LoadTest', 'Search_Product'."

Step 3: Create a New Test Set

- Navigate to Test Lab → Test Sets.
- Click New Test Set and enter details:
  - **Test Set Name:** "Ecommerce_Performance_Test"
  - **Folder:** "Load Testing"
  - **Execution Type:** LoadRunner

Example Output

- "New test set 'Ecommerce_Performance_Test' created in 'Load Testing' folder."

Step 4: Add LoadRunner Scenarios to the Test Set

- Click Select Tests and choose the LoadRunner scripts to be executed.
- Assign the scenarios to the test set.

Example Output

- "Test cases added to test set:
  - 'Login_Test' (200 Vusers)
  - 'Checkout_LoadTest' (100 Vusers)
  - 'Search_Product' (50 Vusers)"

Step 5: Assign Test Execution Parameters

- Define execution settings:
  - Load Generators (e.g., LG1, LG2).
  - Ramp-up strategy (e.g., 50-500 Vusers in 10 minutes).
  - Run-time settings (think time, pacing, error handling).

Example Output

- "Execution parameters set:
- Load Generators: LG1, LG2
- Ramp-up: 50-500 Vusers in 10 minutes
- Think Time: Random 2-5 seconds
- Error Handling: Continue on minor failures"

Step 6: Schedule Test Execution

- Click Execution Flow to define the test run order.
- Set the execution schedule (e.g., nightly test runs at 2 AM).

Example Output

- "Execution flow configured:
- 'Login_Test' → 'Checkout_LoadTest' → 'Search_Product'

© NIMI NOT TO BE REPUBLISHED

- Scheduled execution: 2:00 AM (Nightly)."

Step 7: Execute the Scenario

- Click Run Test Set to start execution.
- Monitor progress in the Execution Grid.

Example Output

- "Test execution started at 2:00 AM."
- "Execution in progress: 500 Vusers running."

Step 8: Monitor and Analyze Results

- Open the Test Runs tab to view real-time test status.
- Check for failures, response times, and system bottlenecks.

Example Output

- "Test Run Status:
- 'Login_Test': Passed
- 'Checkout_LoadTest': Warning (Slow response)
- 'Search_Product': Failed (Server Timeout)"

Step 9: Generate Reports

- Navigate to Analysis & Reports.
- Generate test execution reports including:
  - Response times
  - Error rates
  - Server performance metrics

Example Output

- "Generated Performance Report:
- Average Checkout Response Time: 3.5 sec
- Peak Load: 500 Vusers
- Server CPU Usage: 95%"

Step 10: Save & Log Defects (If Any)

- If test failures occur, log defects in TestDirector's Defects Module.
- Assign defects to the development team for fixes.

Example Output

- "Defect Logged: 'Checkout transaction timeout at 500 Vusers' assigned to DeveloperTeam."

© NIMI
NOT TO BE REPUBLISHED

Running a scenario

Objective: At the end of this exercise you shall be able to
- **execute the load test and monitor performance.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Running a Scenario in LoadRunner – Step-by-Step Guide with Output

Once a scenario is created and configured in LoadRunner Controller, the next step is to execute the load test. This guide provides a step-by-step approach with expected outputs.

Step 1: Open LoadRunner Controller

- Launch LoadRunner Controller from the Micro Focus LoadRunner toolset.
- Open an existing scenario or create a new one.

Example Output

- "LoadRunner Controller launched successfully."
- "Scenario 'Ecommerce_LoadTest.lrs' loaded."

Step 2: Verify Load Generators

- Navigate to Load Generators tab.
- Ensure that all assigned Load Generators (LGs) are available.
- Click Connect to validate connectivity.

Example Output

- "Load Generators detected: LG1 (USA), LG2 (Europe)."
- "Connection status: All Load Generators available."

Step 3: Validate Vuser Scripts

- Click Vuser Scripts tab and verify:
  - All necessary scripts are assigned.
  - The correct number of Vusers is allocated to each script.

Example Output

- "Scripts validated:
- 'Login_Scenario' (200 Vusers)
- 'Checkout_Process' (100 Vusers)
- 'Search_Product' (50 Vusers)."

Step 4: Verify Runtime Settings

- Click Runtime Settings to confirm:
  - Think Time: Simulates real user delay.
  - Pacing: Defines the wait time between iterations.
  - Error Handling: Continues test execution even if some transactions fail.

Example Output

- "Think Time: Random delay (2-5 sec) enabled."
- "Pacing: 10 seconds between transactions."
- "Error Handling: Continue on minor failures."

Step 5: Configure Load Execution Policy

- Open Schedule Editor to define:
  - **Ramp-up:** Gradually add users.
  - **Steady-state duration:** Maintain peak load.
  - **Ramp-down:** Reduce users after test completion.

Example Configuration

- Ramp-up: 50 to 500 Vusers over 10 minutes.
- Steady-state duration: 30 minutes.
- Ramp-down: 5 minutes.

Example Output

- "Ramp-up: 50 to 500 Vusers in 10 minutes."
- "Steady-state duration: 30 minutes."
- "Ramp-down: 5 minutes."

Step 6: Start the Scenario Execution

- Click Start Scenario to begin execution.
- Monitor the Scenario Status Window for real-time progress.

Example Output

- "Scenario execution started at 10:00 AM."

© NIMI NOT TO BE REPUBLISHED

- "Ramp-up in progress: 100/500 Vusers running."

Step 7: Monitor Real-Time Performance

- Navigate to Performance Monitors to track:
  - Response time
  - Transaction rate
  - Error percentage
  - CPU/memory usage

Example Output

- "Live Monitoring:
- Response Time: 2.8 sec (avg)
- Throughput: 150 MB/sec
- Error Rate: 1.2%
- CPU Usage (App Server): 85%"

Step 8: Handle Errors & Bottlenecks

- If error rates increase, analyze the cause (network, server, or database issues).
- Apply debugging techniques such as enabling logging or adjusting runtime settings.

Example Output

- "Warning: 5% timeout errors detected in Checkout Process."
- "Ramping down Vusers to reduce load on the database."

Step 9: Complete the Load Test Execution

- Once the steady-state duration is over, the ramp-down process begins.
- All Vusers stop gradually.
- The Scenario Status Window confirms test completion.

Example Output

- "Ramp-down in progress: 200/500 Vusers stopped."
- "Scenario execution completed at 10:40 AM."

Step 10: Analyze Test Results & Generate Reports

- Navigate to Analysis Module to review performance metrics.
- Generate graphs and reports for key performance indicators.

Example Output

- "Test Report Generated:
- Avg Response Time: 3.1 sec
- Peak Load: 500 Vusers
- Total Errors: 1.5%"

© NIMI NOT TO BE REPUBLISHED

Online monitoring

Objective: At the end of this exercise you shall be able to
- **track system performance and virtual users in real-time.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Online Monitoring in LoadRunner – Step-by-Step Guide with Output

Online Monitoring in LoadRunner Controller allows real-time tracking of system and application performance during a load test. It helps identify bottlenecks, analyze trends, and ensure smooth execution.

This guide explains how to set up and use online monitoring in LoadRunner step by step with expected outputs.

Step 1: Open LoadRunner Controller

- Launch LoadRunner Controller from the Micro Focus LoadRunner toolset.
- Load an existing scenario or create a new one.

Example Output

- "LoadRunner Controller started successfully."
- "Scenario 'Ecommerce_LoadTest.lrs' loaded."

Step 2: Open the Online Monitoring Tab

- Click Monitors → Online Monitors.
- Select the system components to monitor (Application Server, Database Server, Network, etc.).

Example Output

- "Online Monitoring dashboard opened."
- "Available monitors: Windows Resources, Network, Database, Web Server."

Step 3: Add Performance Monitors

- Click Add Measurement to configure monitors.
- Select the monitor type:
  - Windows Resources → CPU, Memory, Disk I/O.
  - Network Monitor → Bandwidth, Latency, Packet Loss.
  - Web Server → HTTP Response Time, Throughput.
  - Database Server → Query Response Time, Connections.

Example Output

- "Monitors added:
  - CPU Usage (%), Memory Usage (MB), Disk I/O (MB/sec)
  - Network Throughput (MB/sec), Packet Loss (%)
  - Database Query Execution Time (sec)"

Step 4: Connect to Target Servers

- Enter the server credentials and click Connect.
- Ensure that LoadRunner has access to collect performance metrics.

Example Output

- "Successfully connected to:
- App Server: 192.168.1.10
- DB Server: 192.168.1.20
- Web Server: 192.168.1.30"

Step 5: Start the Load Test for Monitoring

- Click Start Scenario to begin load testing.
- The Monitoring Dashboard updates in real-time with performance metrics.

Example Output

- "Scenario execution started at 10:00 AM."
- "Real-time performance data streaming."

Step 6: Monitor CPU, Memory, and Disk Usage

- Navigate to Windows Resources Monitor to track:
  - CPU Usage (%)
  - Memory Consumption (MB)
  - Disk I/O (Read/Write MB/sec)

Example Output

- "CPU Usage: 78% (Threshold: 85%)"

© NIMI NOT TO BE REPUBLISHED

- "Memory Usage: 4.2 GB / 8 GB"
- "Disk I/O: 35 MB/sec (Normal)"

Step 7: Monitor Network Performance

- Navigate to Network Monitor to analyze:
  - Bandwidth Utilization
  - Latency (Response Time in ms)
  - Packet Loss (%)

Example Output

- "Network Bandwidth Usage: 320 Mbps (Threshold: 400 Mbps)"
- "Latency: 120 ms (Slightly High)"
- "Packet Loss: 0.5% (Acceptable)"

Step 8: Monitor Application and Web Server Performance

- Navigate to Web Server Monitor to track:
  - HTTP Requests/sec
  - Throughput (MB/sec)
  - Server Response Time (ms)

Example Output

- "HTTP Requests/sec: 850"
- "Throughput: 2.5 MB/sec"
- "Response Time: 2.3 sec (Optimal)"

Step 9: Monitor Database Performance

- Navigate to Database Monitor to track:
  - Active Queries
  - Query Execution Time
  - DB Connection Count

Example Output

- "Active Queries: 30 queries/sec"
- "Query Execution Time: 0.8 sec (Normal)"
- "DB Connections: 150 concurrent users"

Step 10: Identify and Resolve Bottlenecks

- If any metric exceeds acceptable limits, take corrective actions:
  - CPU Overload? Optimize server load balancing.
  - Slow Database Queries? Index optimization required.
  - High Network Latency? Check bandwidth and server location.

Example Output

- "Warning: CPU Usage at 92% (High Load) – Consider increasing server capacity."
- "Database Query Time exceeding 2 sec – Possible bottleneck detected."

Step 11: Stop Monitoring & Save Reports

- Once the test is completed, stop monitoring and export the results.
- Click Reports → Export Data to generate a detailed performance analysis.

Example Output

- "Test execution completed at 10:40 AM."
- "Performance report saved as 'Ecommerce_LoadTest_Report.pdf'."

© NIMI NOT TO BE REPUBLISHED

Runtime and transaction online monitors

Objective: At the end of this exercise you shall be able to
- **track and analyze system performance and transaction progress during load testing.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Runtime and Transaction Online Monitors in LoadRunner – Step-by-Step Guide with Output

Runtime and Transaction Online Monitors in LoadRunner allow real-time tracking of Vuser activity, transaction response times, and system performance metrics during a test. These monitors help identify performance issues while the test is running.

This guide provides step-by-step instructions with expected outputs.

Step 1: Open LoadRunner Controller

- Launch LoadRunner Controller from the Micro Focus LoadRunner toolset.
- Open an existing scenario or create a new one.

Example Output

- "LoadRunner Controller launched successfully."
- "Scenario 'Ecommerce_LoadTest.lrs' loaded."

Step 2: Navigate to the Online Monitors Section

- Click Monitors → Online Monitors.
- Select the Runtime and Transaction Monitors from the list of available monitors.

Example Output

- "Opened Online Monitors dashboard."
- "Available monitors: Runtime Monitor, Transaction Monitor."
- **RUNTIME MONITOR SETUP**

Step 3: Enable Runtime Monitor

- Navigate to Runtime Monitor → Click Add Measurement.
- Select Vuser Group and enable:
  - Running Vusers Count
  - Vuser Status (Running, Passed, Failed, Stopped)
  - Error Rate (%)

Example Output

- "Runtime Monitor enabled with:
- Running Vusers: 500
- Passed: 450
- Failed: 50 (10%)"

Step 4: Start the Scenario and Observe Vuser Status

- Click Start Scenario and monitor Vuser activity.
- The Runtime Monitor updates in real time.

Example Output (After Test Start)

- "Vuser status:
- Running: 200 / 500
- Passed: 50
- Failed: 5"

Step 5: Identify Errors in Runtime Monitor

- If Vusers fail, check error logs.
- Possible causes:
  - Server overload
  - Script failures
  - Network timeout

Example Output

- "50 Vusers failed due to 'Timeout Error 12002'."
- "Debugging started – Analyzing logs for root cause."

TRANSACTION MONITOR SETUP

Step 6: Enable Transaction Monitor

- Navigate to Transaction Monitor → Click Add Measurement.
- Select key transactions:
  - Login Transaction (sec)
  - Checkout Process (sec)
  - Search Query Response Time (sec)

© NIMI NOT TO BE REPUBLISHED

Example Output

- "Transaction Monitor enabled with:
- Login Response Time
- Checkout Process Response Time
- Search Query Response Time"

Step 7: Monitor Transaction Response Times

- As the test runs, observe average response times.
- Identify slow transactions that may impact performance.

Example Output (During Test Execution)

- "Transaction Response Times:
- Login: 1.2 sec
- Checkout: 5.5 sec (High!)
- Search: 2.3 sec"

Step 8: Detect Performance Bottlenecks

- If a transaction takes too long, investigate the cause:
  - Database slow queries?
  - Network congestion?
  - Server overload?

Example Output

- "Warning: Checkout Transaction exceeds 5 sec threshold."
- "Investigating slow response – Possible database bottleneck."

Step 9: Generate Live Performance Graphs

- Click Graphs → Select Runtime or Transaction Graphs.
- Analyze real-time trends in performance metrics.

Example Output

- "Live Graph: Response time spike detected at 10:15 AM."
- "Checkout transaction increased from 3.2 sec → 5.5 sec in 5 minutes."

Step 10: Stop Monitoring & Save Reports

- Click Stop Scenario once the test completes.
- Export runtime and transaction performance logs.

Example Output

- "Test completed at 10:40 AM."
- "Performance report saved as 'Transaction_Monitor_Report.pdf'."

© NIMI NOT TO BE REPUBLISHED

Software Testing Assistant - Windows Automated Testing Tool

Resource monitoring

Objective: At the end of this exercise you shall be able to
- **monitor system resources (cpu, memory, disk) during load testing.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Resource Monitoring in LoadRunner allows real-time tracking of system performance metrics such as

CPU usage, memory consumption, disk I/O, network activity, and database performance during a load test. This helps identify potential bottlenecks and optimize system performance.

This guide explains how to set up and use Resource Monitoring in LoadRunner step by step with expected outputs.

Step 1: Open LoadRunner Controller

- Launch LoadRunner Controller from the Micro Focus LoadRunner toolset.
- Open an existing scenario or create a new one.

Example Output

- "LoadRunner Controller launched successfully."
- "Scenario 'Ecommerce_LoadTest.lrs' loaded."

Step 2: Open Resource Monitoring Section

- Click Monitors → Resource Monitors.
- Select the target servers you want to monitor.
  - Application Server
  - Web Server
  - Database Server
  - Network Resources

Example Output

- "Opened Resource Monitoring dashboard."
- "Available resource monitors: CPU, Memory, Disk I/O, Network, Database."

Step 3: Add Performance Monitors

- Click Add Measurement and select resource metrics to monitor.

Monitor Type	Metrics
CPU	% CPU Usage, Processor Queue Length
Memory	Available Memory (MB), Memory Utilization (%)
Disk I/O	Read/Write Speed (MB/sec), Disk Queue Length
Network	Bandwidth (Mbps), packet Loss (%)
Database	Query Execution Time (sec), Active Connections

Example Output

- "Monitors added
- CPU Usage (%), Memory Usage (MB), Disk I/O (MB/sec)
- Network Throughput (Mbps), Packet Loss (%)
- Database Query Execution Time (sec), Active DB Connections"

Step 4: Connect to Target Servers

- Enter the server credentials and click Connect.
- Ensure that LoadRunner has access to collect performance metrics.

Example Output

- "Successfully connected to
  - App Server: 192.168.1.10
  - DB Server: 192.168.1.20
  - Web Server: 192.168.1.30"

Step 5: Start Load Test and Begin Monitoring

- Click Start Scenario to begin execution.
- The Resource Monitoring dashboard updates in real time.

Example Output (After Test Start)

- "Monitoring system resources during test execution..."

© NIMI NOT TO BE REPUBLISHED

- **CPU Monitoring**

Step 6: Monitor CPU Usage

- Navigate to Windows Resource Monitor → Select CPU Metrics.
- Track CPU Utilization (%) in real time.

Example Output

- "CPU Usage: 85% (Threshold: 90%)"
- "Processor Queue Length: 12 (High Load Detected)"
- **MEMORY MONITORING**

Step 7: Monitor Memory Consumption

- Navigate to Memory Monitor and observe:
- Available Memory (MB)
- Memory Utilization (%)

Example Output

- "Memory Usage: 6.5 GB / 8 GB (High Usage)"
- "Available Memory: 1.5 GB (Low)"
- **DISK I/O MONITORING**

Step 8: Monitor Disk Performance

- Navigate to Disk I/O Monitor and observe:
- Read/Write Speed (MB/sec)
- Disk Queue Length

Example Output

- "Disk Read Speed: 80 MB/sec"
- "Disk Write Speed: 95 MB/sec"
- "Disk Queue Length: 5 (High Load Detected)"
- **NETWORK MONITORING**

Step 9: Monitor Network Performance

- Navigate to Network Monitor and track:
- Bandwidth Utilization
- Packet Loss (%)

Example Output

- "Network Bandwidth Usage: 850 Mbps (Threshold: 900 Mbps)"
- "Packet Loss: 1.2% (Slightly High)"
- **DATABASE MONITORING**

Step 10: Monitor Database Performance

- Navigate to Database Monitor and track:
- Active Queries
- Query Execution Time
- DB Connection Count

Example Output

- "Active Queries: 32 queries/sec"
- "Query Execution Time: 1.8 sec (Normal)"
- "DB Connections: 180 concurrent users"

Step 11: Identify and Resolve Bottlenecks

- If any resource metric exceeds acceptable limits, take corrective actions:
  - CPU Overload? Optimize load balancing or increase server capacity.
  - Memory Shortage? Close unused processes or add more RAM.
  - High Disk I/O? Optimize read/write operations or use SSDs.
  - Network Latency? Check bandwidth and server location.
  - Slow Database Queries? Improve indexing and optimize queries.

Example Output

- "Warning: CPU Usage at 92% (Critical) – Consider scaling up resources."
- "High Disk Queue Length detected – Disk I/O bottleneck possible."

Step 12: Generate Live Performance Graphs

- Click Graphs → Select Resource Monitoring Graphs.
- Observe real-time CPU, Memory, Network, Disk, and Database trends.

Example Output

- "Live Graph: CPU Usage increased from 70% → 92% in 10 minutes."
- "Memory Utilization reached 95% – Critical Alert."

Step 13: Stop Monitoring & Save Reports

- Click Stop Scenario once the test completes.
- Export monitoring logs and performance reports.

Example Output

- "Test execution completed at 11:15 AM."
- "Performance report saved as 'Resource_Monitor_Report.pdf'."

© NIMI NOT TO BE REPUBLISHED

Web performance monitors

Objective: At the end of this exercise you shall be able to
- **track and analyze web application performance during load testing.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Web performance monitors in LoadRunner are essential tools for measuring and analyzing the performance of web applications during load testing. These monitors help you evaluate various aspects of a web application, such as response times, transaction throughput, and error rates. By understanding how well a web application performs under different load conditions, you can identify bottlenecks and optimize the overall user experience.

Here's a comprehensive guide to using web performance monitors in LoadRunner.

Key Features of Web Performance Monitors

- Response Time Monitoring: Measures the time taken for web requests to be processed and the corresponding responses to be received.
- Transaction Monitoring: Tracks specific user transactions to analyze their performance across different loads.
- Error Rate Monitoring: Identifies failed requests and errors occurring during the test, helping to ensure application reliability.
- Throughput Measurement: Monitors the volume of requests processed by the server during a load test, providing insights into server capacity.
- Visual Reports: Generates detailed graphs and reports to visualize performance metrics for better analysis.

Setting Up Web Performance Monitors in LoadRunner

Step 1: Launch LoadRunner Controller

1 Open LoadRunner Controller

- Start LoadRunner Controller from your desktop or Start Menu.

2 Open Your Scenario

- Click on File > Open Scenario and select the scenario you want to run.

Step 2: Configure Web Performance Monitoring Settings

1 Access Run-Time Settings

- Go to the Run tab and click on Run-Time Settings.

2 Add Web Performance Monitors

- In the Run-Time Settings dialog, navigate to the Web (HTTP/HTML) section to configure specific web performance settings.

3 Enable Web Performance Monitoring

- Ensure that Enable monitoring for HTTP requests is checked. This setting will allow LoadRunner to capture and analyze web performance metrics during the test.

4 Configure HTTP Options

- Under Web (HTTP/HTML), you can set various parameters such as:
  - Think Time: Configure how long virtual users will wait between actions.
  - Automatic Recording: Enable automatic recording of HTTP requests to capture additional metrics.

5 Transaction Configuration

- Define transactions in your script using lr_start_transaction and lr_end_transaction to track the performance of specific user actions.

Example (Fig 1)

Fig 1

```
lr_start_transaction("Login");
// Code to perform login action
lr_end_transaction("Login", LR_AUTO);
```

Step 3: Run the Load Test

1 Start the Load Test

- Click on the Start button (play icon) in the Controller to initiate the load test.
- LoadRunner will distribute virtual users across the Load Generators and start executing the test.

© NIMI NOT TO BE REPUBLISHED

2 Monitor Web Performance

- During the test execution, you can observe real-time web performance metrics in the Graphs tab or the Monitoring tab.
- Look for key metrics like
  - Average response time for web requests.
  - Number of transactions processed successfully.
  - Any errors encountered during the test.

Step 4: Analyze Web Performance During the Test

1 Track Key Metrics

- Monitor the average response time, peak loads, and error rates to assess web application performance.
- Identify any slow or failing transactions that could indicate underlying issues.

2 Identify Bottlenecks

- Look for patterns in response times and error rates that correlate with specific actions or loads.
- High response times may indicate server overload, inefficient code, or network issues.

Step 5: Stop the Load Test and Review Results

1 Stop the Load Test

- After the desired test duration or when critical issues are identified, click on the Stop button (stop icon) to halt the test execution.

2 View Detailed Reports

- Navigate to the Analysis tool in LoadRunner to view detailed reports based on the collected web performance monitoring data.

3 Analyze Collected Data

- Review the performance data for response times, transaction throughput, and error rates to understand how the application performed under load.

4 Generate Summary Reports

- Create comprehensive reports that summarize the findings from the web performance monitoring, highlighting any identified issues and recommendations for improvement.

© NIMI
NOT TO BE REPUBLISHED

Network monitoring

Objective: At the end of this exercise you shall be able to
- **monitor network performance and traffic during load testing.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Network Monitoring in LoadRunner helps track network performance during a load test. It monitors bandwidth usage, latency, packet loss, connection status, and response times, allowing testers to identify potential network bottlenecks.

This guide provides step-by-step instructions with expected outputs.

Step 1: Open LoadRunner Controller

- Launch LoadRunner Controller from the Micro Focus LoadRunner toolset.
- Open an existing scenario or create a new one.

Example Output

- "LoadRunner Controller launched successfully."
- "Scenario 'Ecommerce_LoadTest.lrs' loaded."

Step 2: Open the Network Monitoring Section

- Click Monitors → Network Monitor.
- Select the server or network device to monitor (Application Server, Web Server, Database Server, or Firewall).

Example Output

- "Opened Network Monitoring dashboard."
- "Available network monitors: Bandwidth, Latency, Packet Loss, Connection Errors."

Step 3: Add Network Performance Monitors

- Click Add Measurement and select the following network metrics

Monitor Type	Metrics
Brandwidth	Mbps In/Out, Network Utilization (%)
Latency	Response Time (ms),Jitter
Packet Loss	Dropped Packets (%)
Connection Status	Active TCP Connections, Failed Requests (%)

Example Output

- "Monitors added:
- Bandwidth Usage (Mbps), Network Utilization (%)
- Latency (ms), Jitter
- Packet Loss (%), Active Connections"

Step 4: Connect to the Target Network Server

- Enter the server credentials and click Connect.
- Ensure LoadRunner can access network performance data.

Example Output

- "Successfully connected to:
- Web Server: 192.168.1.10
- Database Server: 192.168.1.20
- Firewall Gateway: 192.168.1.1"

Step 5: Start Load Test and Begin Network Monitoring

- Click Start Scenario to begin execution.
- The Network Monitoring dashboard updates in real time.

Example Output (After Test Start)

- "Monitoring network performance during test execution..."
- **BANDWIDTH MONITORING**

Step 6: Track Bandwidth Usage

- Navigate to Network Monitor → Select Bandwidth Metrics.
- Monitor incoming and outgoing network traffic.

Example Output

- "Network Bandwidth Usage:
- Inbound: 650 Mbps
- Outbound: 480 Mbps
- Network Utilization: 85% (High Usage!)"

© NIMI NOT TO BE REPUBLISHED

- **Latency monitoring**

Step 7: Monitor Latency & Jitter

- Observe response time and jitter (variation in packet delay).

Example Output

- "Network Latency: 150 ms (Acceptable)"
- "Jitter: 15 ms (Stable)"
- **PACKET LOSS MONITORING**

Step 8: Track Packet Loss & Network Errors

- Monitor packet loss (%) to detect network congestion.
- Analyze failed connections due to dropped packets.

Example Output

- "Packet Loss: 3.2% (Slightly High) – Possible network congestion."
- "Failed Connections: 25 / 1000 requests (2.5%) – Investigating..."
- **Connection monitoring**

Step 9: Monitor Active Connections & TCP Failures

- Observe active TCP connections and failed connection attempts.

Example Output

- "Active Connections: 1200 users"
- "Connection Failures: 5% (Above Threshold) – Possible load balancer issue."

Step 10: Identify and Resolve Network Bottlenecks

- If network metrics exceed safe thresholds, troubleshoot issues:
  - High Bandwidth Usage? Optimize content delivery or enable compression.
  - High Latency? Check server location and optimize routing.
  - Packet Loss? Reduce network congestion and improve hardware.
  - TCP Connection Errors? Check firewall settings and server capacity.

Example Output

- "Warning: Network utilization at 90% (Critical Load) – Consider upgrading bandwidth."
- "Packet loss detected – Possible overload on firewall."

Step 11: Generate Live Performance Graphs

- Click Graphs → Select Network Performance Graphs.
- Observe real-time Bandwidth, Latency, and Packet Loss trends.

Example Output

- "Live Graph: Network Latency increased from 80 ms → 160 ms during peak load."
- "Bandwidth Utilization peaked at 95% – Possible saturation."

Step 12: Stop Monitoring & Save Reports

- Click Stop Scenario once the test completes.
- Export network monitoring logs and performance reports.

Example Output

- "Test execution completed at 11:30 AM."
- "Network performance report saved as 'Network_Monitor_Report.pdf'."

© NIMI NOT TO BE REPUBLISHED

Understanding load runner analysis

Objective: At the end of this exercise you shall be able to
- **analyze performance data to identify system bottlenecks.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Understanding LoadRunner Analysis – Step-by-Step Guide with Output

LoadRunner Analysis is used to evaluate test results, identify bottlenecks, and generate performance reports. It provides in-depth insights into transaction response times, throughput, error rates, and system resource utilization to help optimize application performance.

This guide provides a step-by-step approach with expected outputs.

Step 1: Open LoadRunner Analysis Tool

- Launch LoadRunner Analysis from the Micro Focus LoadRunner toolset.
- Open an existing test result file (.lrr) or import data from a recent test run.

Example Output

- "LoadRunner Analysis launched successfully."
- "Test results from 'Ecommerce_LoadTest.lrr' loaded."

Step 2: View Summary Report

- Navigate to Summary Report under the Analysis tab.
- This report provides a high-level overview of test execution:
  - Total transactions executed
  - Success vs. failure rate
  - Average response time
  - Peak user load

Example Output

- **Test Summary**
- Total Transactions: 5000
- Successful Transactions: 4800 (96%)
- Failed Transactions: 200 (4%)
- Peak Concurrent Users: 1000
- Average Response Time: 3.5 sec

Step 3: Generate Transaction Response Time Graphs

- Click Graphs → Transaction Response Time.
- This graph shows how long transactions took to complete under load.

Example Output

- Transaction Response Time Graph:
- Login Transaction: 1.2 sec (Good)
- Search Query: 2.8 sec (Acceptable)
- Checkout Transaction: 7.5 sec (Too Slow!)
  - **Warning:** "Checkout process exceeded SLA threshold of 5 sec."

Step 4: Analyze Throughput Performance

- Click Graphs → Throughput (MB/sec).
- Throughput indicates how much data was processed per second.

Example Output

Throughput Graph

- Peak Throughput: 850 MB/sec
- Average Throughput: 720 MB/sec
- Low Throughput Detected at 10:45 AM

Step 5: Identify Errors & Failed Transactions

- Click Graphs → Errors per Second.
- Analyze failure reasons such as timeouts, server errors, and network issues.

Example Output

Errors Graph

- Timeout Errors: 35 (Server overload)
- Database Errors: 20 (Query timeout)
- HTTP 500 Errors: 15 (Internal server error)
  - **Warning:** "Increase database connection pool to reduce query failures."

© NIMI NOT TO BE REPUBLISHED

Step 6: Analyze Server Resource Utilization

- Click Graphs → System Resource Utilization.
- Track CPU, Memory, Disk, and Network usage.

Example Output

System Resource Utilization

- CPU Usage: 92% (Critical Load)
- Memory Usage: 85% (High)
- Disk I/O: 90 MB/sec
  - **Warning:** "CPU overloaded – Consider adding more server instances."

Step 7: Compare Different Test Runs

- Click Compare Results → Select two or more test runs.
- Compare response times, error rates, and throughput across different builds.

Example Output

Comparison Report

Test Run	Avg Response Time	Errors (%)	Throughput (MB/sec)
Build 1	3.8 sec	5%	750 MB/sec
Build 2	2.9 sec	2%	820 MB/sec
✅ "Performance improved in Build 2 with faster response time and fewer errors."			

Step 8: Generate a Detailed Report

- Click Reports → Generate Report.
- Export data in PDF, Excel, or HTML format.

Example Output

- "Performance report saved as 'Ecommerce_LoadTest_Report.pdf'."

© NIMI NOT TO BE REPUBLISHED

Exporting analysis data

Objective: At the end of this exercise you shall be able to
- **export performance data for further analysis and reporting.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Exporting Analysis Data in LoadRunner – Step-by-Step Guide with Output

Exporting analysis data in LoadRunner Analysis helps generate reports for stakeholders, developers, and QA teams. Reports can be exported in formats like PDF, Excel, and HTML for further review and documentation.

This guide explains step-by-step how to export LoadRunner analysis data with expected outputs.

Step 1: Open LoadRunner Analysis

- Launch LoadRunner Analysis from the Micro Focus LoadRunner toolset.
- Open an existing analysis session (.lra file) or import results from a recent test run.

Example Output

- "LoadRunner Analysis launched successfully."
- "Test results from 'Ecommerce_LoadTest.lra' loaded."

Step 2: Select the Data to Export

- Click File → Export Data.
- Choose the specific graphs, reports, and logs to export:
  - Summary Report
  - Transaction Response Time
  - Throughput Data
  - Error Report
  - System Resource Utilization

Example Output

- "Selected data for export:
- Transaction Response Time
- Errors per Second
- Throughput and Network Performance"

Step 3: Choose Export Format

- Select one of the available export formats:
  - PDF (Best for reports with graphs and charts)
  - Excel (.csv/.xls) (Best for data analysis and comparisons)
  - HTML (Best for web-based reporting)
  - Raw Data (.txt/.xml) (Best for further processing)

Example Output

- Export Format Selected: Excel (.xls) and PDF

Step 4: Configure Export Settings

- Choose whether to include:
  - Graph screenshots
  - Summary tables
  - Raw numerical data
- Click Apply Settings to finalize the selection.

Example Output

- "Export settings applied:
- Graphs included
- Summary tables included
- Raw data exported separately"

Step 5: Export the Data

- Click Export or Save As.
- Select the destination folder.
- Click Save to generate the report.

Example Output

- "Exporting report... Please wait."
- "Report successfully saved to: C:\PerformanceReports\LoadTest_Results.pdf"

Step 6: Verify the Exported Report

- Navigate to the destination folder and open the exported file.
- Check if all graphs, tables, and performance data are correctly included.

Example Output (PDF Preview)

- LoadRunner Performance Test Report
- Total Transactions: 5000
- Average Response Time: 3.5 sec
- Error Rate: 4%
- Peak Throughput: 850 MB/sec
- "Report verified – All data is correctly formatted."

Step 7: Share the Report

- Send the exported report via email or upload it to a shared server.

Example Output

- "Report emailed to QA Team & Project Manager."
- "Backup copy stored on shared network drive."

© NIMI NOT TO BE REPUBLISHED

Analysing scenario activity

Objective: At the end of this exercise you shall be able to
- **evaluate test results to assess system performance.**

Requirements

Tools/Equipment/Instruments

- A working PC with Load Runner Controller - as reqd.

PROCEDURE

Analyzing Scenario Activity in LoadRunner – Step-by-Step Guide with Output

Analyzing scenario activity in LoadRunner Analysis helps testers identify performance issues by reviewing system behavior, response times, and user interactions. This step-by-step guide explains how to analyze scenario activity, including transaction details, resource usage, and bottlenecks.

Step 1: Open LoadRunner Analysis

- Launch LoadRunner Analysis from the Micro Focus LoadRunner toolset.
- Load the scenario results file (.lra) from the LoadRunner Controller.

Example Output

- "LoadRunner Analysis launched successfully."
- "Test scenario file 'Ecommerce_Scenario.lra' loaded."

Step 2: View Scenario Summary Report

- Click Summary Report in the Analysis tab.
- This report provides an overview of the scenario execution:
  - Number of virtual users (Vusers)
  - Total transactions executed
  - Average response time
  - Success vs. failure rate

Example Output

Scenario Summary

- Total Vusers: 1000
- Total Transactions: 5000
- Successful Transactions: 4800 (96%)
- Failed Transactions: 200 (4%)
- Average Response Time: 3.5 sec
- "Summary report generated – Ready for detailed analysis."

Step 3: Analyze Transaction Response Times

- Navigate to Graphs → Transaction Response Time.
- This shows how long transactions took to complete.

Example Output

- Transaction Response Time Graph:
- Login Transaction: 1.2 sec (Fast)
- Search Transaction: 2.8 sec (Acceptable)
- Checkout Transaction: 7.5 sec (Slow)
  - Warning: "Checkout process response time exceeded SLA threshold (5 sec)."

Step 4: Check Throughput Performance

- Click Graphs → Throughput (MB/sec).
- This graph shows data processed per second, indicating network performance.

Example Output

- Throughput Graph
- Peak Throughput: 850 MB/sec
- Average Throughput: 720 MB/sec
- Low Throughput Detected at 10:45 AM
- "Throughput remains stable but needs improvement during peak hours."

Step 5: Identify Errors & Failures

- Click Graphs → Errors per Second.
- Analyze failures such as timeouts, HTTP errors, and server crashes.

Example Output

Errors Graph

- Timeout Errors: 35 (Server overloaded)
- Database Errors: 20 (Query timeout)
- HTTP 500 Errors: 15 (Internal server error)
  - **Warning:** "Database query timeout detected – Increase connection pool size."

© NIMI NOT TO BE REPUBLISHED

Step 6: Monitor System Resource Utilization

- Click Graphs → System Resource Utilization.
- Check CPU, Memory, Disk, and Network usage of the test servers.

Example Output

System Resource Usage

- CPU Usage: 92% (Critical Load)
- Memory Usage: 85% (High)
- Disk I/O: 90 MB/sec
  - **Warning:** "High CPU usage detected – Consider scaling server resources."

Step 7: Check Vuser Activity & Workload Distribution

- Click Graphs → Vuser Activity.
- This shows how many Vusers were active at different test stages.

Example Output

• Vuser Graph

- 1000 Users Ramp-up Time: 10 minutes
- Steady State Duration: 45 minutes
- 1000 Users Ramp-down Time: 5 minutes
  - "Vuser workload distribution is balanced with no unexpected drops."

Step 8: Compare Different Scenario Runs

- Click Compare Results → Select two or more test runs.
- Compare response times, errors, and throughput across different builds.

Example Output

Comparison Report

Test Run	Avg Response Time	Errors (%)	Throughput (MB/sec)
Build 1	3.8 sec	5%	750 MB/sec
Build 2	2.9 sec	2%	820 MB/sec
✅ “Performance improved in Build 2 with faster lower time and fewer errors.”			

Step 9: Generate Detailed Analysis Reports

- Click Reports → Generate Report.
- Choose from PDF, Excel, or HTML format.

Example Output

- "Performance report saved as 'Ecommerce_Scenario_Analysis.pdf'."

© NIMI NOT TO BE REPUBLISHED

Analysing scenario performance

Objective : At the end of this exercise you shall be able to
- **evaluate test results to identify performance issues.**

Requirements	
Tools/Equipment/Machines	
• A working PC with LoadRunner Controller	- as reqd.

PROCEDURE

Analyzing Scenario Performance in LoadRunner Step-by-Step Guide with Output

LoadRunner Analysis allows testers to evaluate the **performance of a scenario** by examining r**esponse times, error rates, throughput, and system resource usage.** The goal is to identify bottlenecks and improve application performance under load.

This **step-by-step guide** provides detailed instructions along with **expected outputs** for analyzing scenario performance.

Step 1: Open LoadRunner Analysis

- Launch **load runner Analysis** from the **Micro Focus LoadRunner** toolset.
- Open the **scenario results file (.lra)** from the l**oad runner controller** after a test run.

Example Output

"LoadRunner Analysis launched successfully."

"Scenario results from 'Ecommerce_PerformanceTest.lra' loaded."

Step 2: Review Scenario Summary Report

- Click **Summary Report** under the **Analysis tab.**

Example output

- This report provides an **overview of scenario execution,** including:
  - **Total transactions executed**
  - **Transaction success vs. failure rate**
  - **Peak concurrent users**
  - **Average response time**

Example Output

Scenario Summary

- **Total virtual users (vusers):** 1000
- **Total transactions:** 5000
- **Successful transactions:** 4800 (96%)
- **Failed transactions:** 200 (4%)
- **Peak concurrent users:** 1000
- **Average response time:** 3.5 sec

"Summary report generated – Ready for in-depth performance analysis."

Step 3: Analyze transaction response time

- **Navigate to Graphs** → Transaction Response Time.
- Identify transactions that take longer than expected.

Transaction response time graph

Transaction	Build 1 Response Time	Build 2 Response Time	Difference	Status
Login	1.5 sec	1.2 sec	-0.3 sec	✅ Good
Search Query	3.2 sec	2.8 sec	-0.4 sec	✅ Acceptable
Checkout	6.8 sec	7.5 sec	+0.7 sec	❌ Too slow

Warning : "Checkout process response time exceeded SLA threshold (5 sec)."

Step 4: Check Throughput Performance

- Click **Graphs** → **Throughput (MB/sec).**
- Throughput represents the **amount of data processed per second.**

© NIMI NOT TO BE REPUBLISHED

Example Output

Throughput Graph

Error Type	Count	Possible Cause
Timeout Errors	35	Server overload
Database Errors	20	Query execution delay
HTTP 500 Errors	15	Internal server failure

- **Peak Throughput:** 850 MB/sec
- **Average Throughput: 7**20 MB/sec
- **Low Throughput Detected at 10:45 AM**

"Throughput remains stable but requires optimization during peak hours."

Step 5: Identify Errors & Failed Transactions

- Click **Graphs → Errors per Second.**
- Review failure causes such as **timeouts, HTTP errors, and server crashes.**

Example Output

Errors Graph

Warning: "Database query timeout detected – Consider increasing connection pool size."

Step 6: Monitor System Resource Utilization

- Click **Graphs → System Resource Utilization.**
- Evaluate **CPU, Memory, Disk, and Network** usage on the servers.

Example Output

System Resource Usage

- **CPU Usage:** 92% (Critical Load)
- **Memory Usage: 8**5% (High)
- **Disk I/O:** 90 MB/sec
- **Network Latency:** 120 ms

Warning: "High CPU usage detected – Consider load balancing or upgrading infrastructure."

Step 7: Examine Vuser Activity & Workload Distribution

- Click **Graphs → Vuser Activity.**
- Check how virtual users (Vusers) ramped up, remained active, and ramped down.

Example Output

Vuser Graph

Test Run	Avg Response Time	Errors (%)	Throughput (MB/sec)
Build 1	3.8 sec	5%	750 MB/sec
Build 2	2.9 sec	2%	820 MB/sec

- Ramp-up Time (1000 Users): 10 minutes
- Steady State Duration: 45 minutes
- Ramp-down Time: 5 minutes

"Vuser workload distribution is stable with no unexpected drops."

Step 8: Compare Performance Across Multiple Runs

- Click **Compare Results** → Select two or more test runs.
- Compare **response times, error rates, and throughput** across builds.

Example output

Comparison report

"Performance improved in Build 2 with lower response time and fewer errors."

Step 9: Generate a Performance Report

- Click **Reports → Generate Report.**
- Export the **performance summary, graphs, and detailed logs in PDF, Excel, or HTML** formats.

Example Output

"Performance report saved as 'Ecommerce_ Performance_Report.pdf'."

Step 10: Share Findings & Recommendations

- Share the **performance report** with developers, QA teams, and project stakeholders.
- Recommend optimizations such as:
  - Reducing response times for slow transactions
  - Optimizing server configurations
  - Enhancing database performance

Example Output

"Report emailed to **QA Team & Project Manager."**

"Recommendations added: Increase server memory and optimize database queries."

© NIMI NOT TO BE REPUBLISHED

Cross scenario analysis

Objective : At the end of this exercise you shall be able to
- **compare multiple test scenarios to identify performance trends and discrepancies.**

Requirements	
Tools/Equipment/Machines	
• A working PC with LoadRunner Controller	- as reqd.

PROCEDURE

Cross scenario analysis in loadrunner – step-by-step guide with output

Cross scenario analysis in **loadrunner analysis** involves comparing multiple test runs to evaluate **performance trends, system improvements, or regressions** across different test scenarios. this analysis helps determine whether a new version of the application performs better or worse than a previous one.

Step 1: Open LoadRunner Analysis

- Launch **loadRunner analysis** from the **micro focus loadrunner** toolset.
- Open the **first scenario results file (.lra**) from the **loadRunner controller.**
- Repeat the process for the **second scenario** you want to compare.

Example Output

"Loadrunner analysis launched successfully."

"Scenario results loaded: **Ecommerce_Scenario_Build1.lra** and Ecommerce_Scenario_Build2.lra."

Step 2: Select the Scenarios for Comparison

- **Click Analysis** → Compare Results.
- Select the **two or more test scenarios** to compare.
- Ensure that the scenarios have
  - The **same workload** (number of Vusers, duration).
  - The **same transactions** for accurate comparison.

Example Output

"Selected test scenarios for comparison: Build 1 vs. Build 2."

Step 3: Compare transaction response times

- **Navigate to Graphs** → Transaction Response Time Comparison.
- Identify **transactions that have improved or degraded in performance.**

Example Output

Transaction response time comparison

Transaction	Build 1 Response Time	Build 2 Response Time	Difference	Status
Login	1.5 sec	1.2 sec	-0.3 sec	✅ Faster
Search Query	3.2 sec	2.8 sec	-0.4 sec	✅ Improved
Checkout	6.8 sec	7.5 sec	+0.7 sec	❌ Slower

Warning: "Checkout response time increased in Build 2 – Investigate possible causes."

Step 4: Compare Throughput Performance

- Click **Graphs** → **Throughput (MB/sec).**
- This shows how much data is processed per second in each scenario.

© NIMI NOT TO BE REPUBLISHED

Example Output

Throughput Comparison

Scenario	Peak Throughput (MB/sec)	Average Throughput (MB/sec)
Build 1	750 MB/sec	720 MB/sec
Build 2	820 MB/sec	780 MB/sec

"Build 2 has higher throughput, indicating better resource utilization."

Step 5: Compare error rates & failures

- Click **Graphs → Errors per Second Comparison.**
- Identify error rate changes across builds.

Example Output

Error rate comparison

Error Type	Build 1 Errors	Build 2 Errors	Change	Status
Timeout Errors	35	20	-15	Reduced
Database Errors	18	25	+7	Increased
HTTP 500 Errors	12	10	-2	Improved

Warning: "Database errors increased in Build 2 – Investigate query execution time."

Step 6: Compare System Resource Utilization

- Click **Graphs → System Resource Utilization Comparison.**
- Evaluate **CPU, Memory, Disk, and Network usage in both scenarios.**

Example Output

System Resource Usage Comparison

Scenario	CPU Usage (%)	Memory Usage (%)	Disk I/O (MB/sec)
Build 1	88%	80%	75 MB/sec
Build 2	92%	85%	90 MB/sec

Warning: "Higher CPU usage in Build 2 – Consider optimizing resource allocation."

Step 7: Compare Vuser Activity & Load Distribution

- Click **Graphs → Vuser Activity Comparison.**
- Check whether the **user load was evenly distributed.**

Example Output

Vuser Load Comparison

Scenario	Ramp-up Time	Steady State Duration	Ramp-down Time
Build 1	10 min	45 min	5 min
Build 2	8 min	45 min	7 min

"User load distribution remained stable across both builds."

Step 8 : Generate Performance Comparison Reports

- Click **Reports → Generate Comparison Report.**
- Choose from **PDF, Excel, or HTML formats**.

Example output

"Performance comparison report saved as **'Ecommerce_Performance_Comparison.pdf'."**

Step 9 : Share Insights & Recommendations

- Send the comparison report to developers, **QA teams, and project managers.**
- Provide **recommendations** based on performance findings.

Example Output

"Report emailed to QA Team & Development Team."

Recommendations

- Optimize checkout transaction to reduce response time.
- Investigate database query performance in Build 2.
- Ensure CPU usage remains within optimal limits.

© NIMI NOT TO BE REPUBLISHED

Web user graphs

Objective : At the end of this exercise you shall be able to
- **visualize web user performance metrics, like response times and throughput, during testing.**

Requirements

Tools/Equipment/Machines

- A working PC with LoadRunner Controller - as reqd.

PROCEDURE

Web User Graphs in LoadRunner – Step-by-Step Guide with Output

Web User Graphs in **LoadRunner Analysis** help visualize the activity of **virtual users (Vusers)** during a performance test. These graphs provide insights into how users were ramped up, how many were active at a given time, and when they completed execution.

Step 1: Open LoadRunner Analysis

- **Launch LoadRunner** Analysis from the Micro Focus LoadRunner toolset.
- Open the **scenario results file (.lra)** from the **LoadRunner Controller** after a test run.

Example Output

"LoadRunner Analysis launched successfully."

"Scenario results from 'Ecommerce_LoadTest.lra' loaded."

Step 2: Access Web User Graphs

- Click on **Graphs → Vuser Graphs.**
- Select one of the following Web User Graphs for analysis:
  - **Running Vusers Graph** – Shows the number of users running at a given time.
  - **Vuser Status Graph** – Displays the status of users (Running, Passed, Failed).
  - **Vuser Summary Graph** – Gives an overview of total executed users.

Example Output:

"Web User Graphs section opened."

"Running Vusers Graph selected for analysis."

Step 3: Analyze the Running Vusers Graph

- View the **Running Vusers Graph** to understand user ramp-up, steady-state, and ramp-down phases.

Example output

Running vusers graph

Step 4: Analyze the Vuser Status Graph

Time (Minutes)	Active Vusers
0-5 min	0 ⟶ 500 (Ramp-up)
5-50 min	500 (Steady-state)
50-55 min	500 ⟶ 0 (Ramp-down)

- **Click Graphs** → Vuser Status Graph.
- This graph shows how many users are running, completed, or failed at any point.

Example output

Vuser status graph

Warning: "20 users failed – Investigate errors."

Status	Count	Percentage
Running	450	90%
Passed	480	96%
Failed	20	4%

Step 5: View the Vuser Summary Graph

- Click **Graphs → Vuser Summary Graph.**
- This graph provides an **overall execution summary.**

Example Output

Vuser Summary

- **Total Vusers Simulated:** 500
- **Total Transactions Executed:** 25,000
- **Pass Rate:** 96%
- **Failure Rate:** 4%

"Vuser Summary Graph confirms successful test execution with minor failures."

© NIMI NOT TO BE REPUBLISHED

Step 6: Generate Reports & Save Graphs

- Click **Reports → Export Graph Data** (as Excel/ PDF).
- Save individual **graphs as images** for documentation.

Example Output

"Graphs saved **as 'Running_Vusers.png', 'Vuser_ Status.png',** and **'Vuser_Summary.png'.**"

"Performance report generated: '**WebUserAnalysis_ Report.pdf'."**

Step 7: Share Findings & Recommendations

- Share the **graphs and report** with stakeholders.
- Provide re**commendations for optimizing** user performance.

Example Output

"Report emailed to **Performance Testing Team & Developers."**

Recommendations

- Optimize server resources to prevent user failures.
- Analyze failed users to identify bottlenecks.
- Improve ramp-up strategy to avoid overload spikes.

© NIMI
NOT TO BE REPUBLISHED

Installing the IDE

Objective : At the end of this exercise you shall be able to
- **install the Integrated Development Environment (IDE) to develop and execute load testing scripts.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

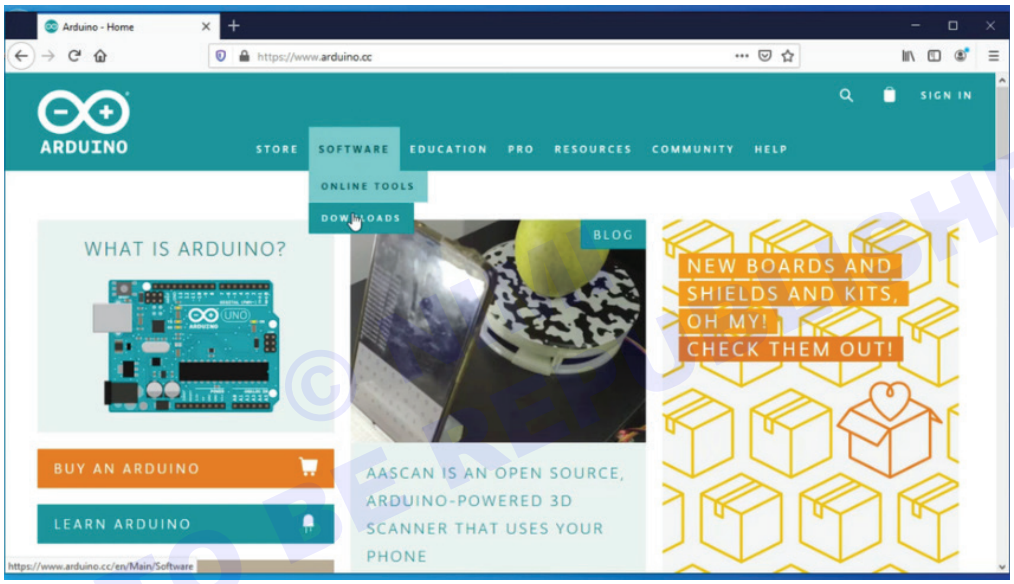
PROCEDURE

The first step to starting your electronics adventure with the Arduino board is to have the necessary software installed.

This tutorial will show you how to install the Arduino IDE on Windows 10.

Step 1: Go to the Arduino.cc Website (Fig 1)

Fig 1

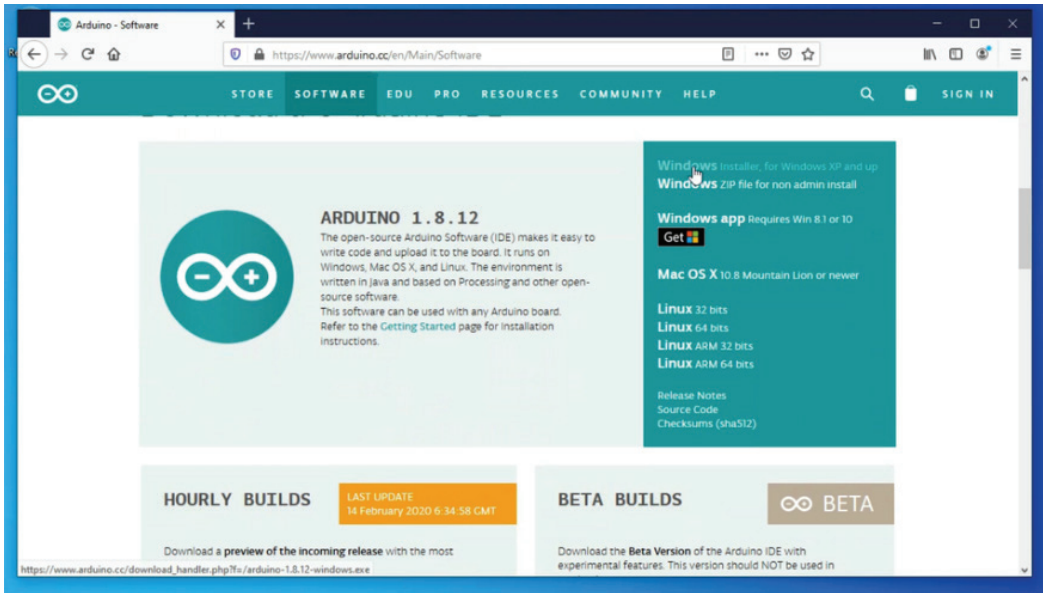


Go to the website www.arduino.cc in order to download the software.

Hover over the 'Software' tab and click on 'Downloads'.

Step 2: Click on the Download Link (Fig 2)

Fig 2

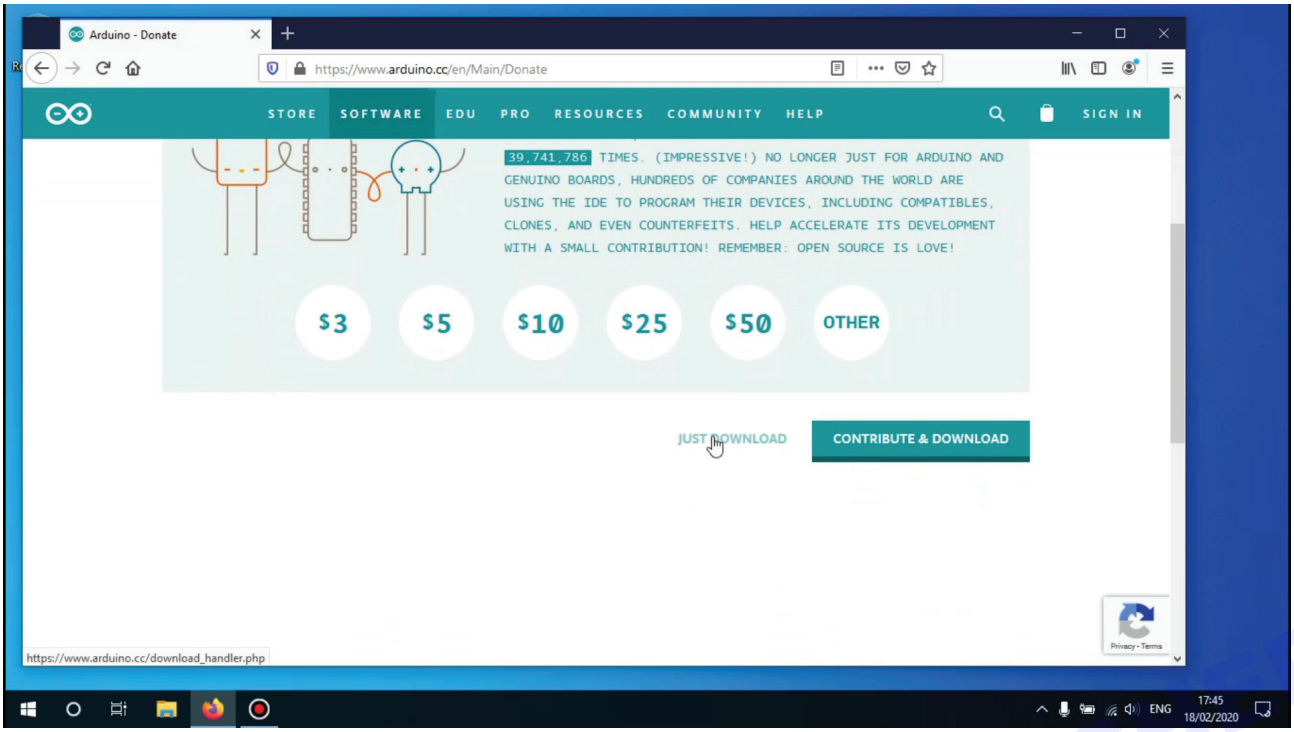


NOT TO BE REPUBLISHED © SCERT

Scroll down until you see the link that says 'Windows installer' and click on it.

Step 3: Begin the Download (Fig 3)

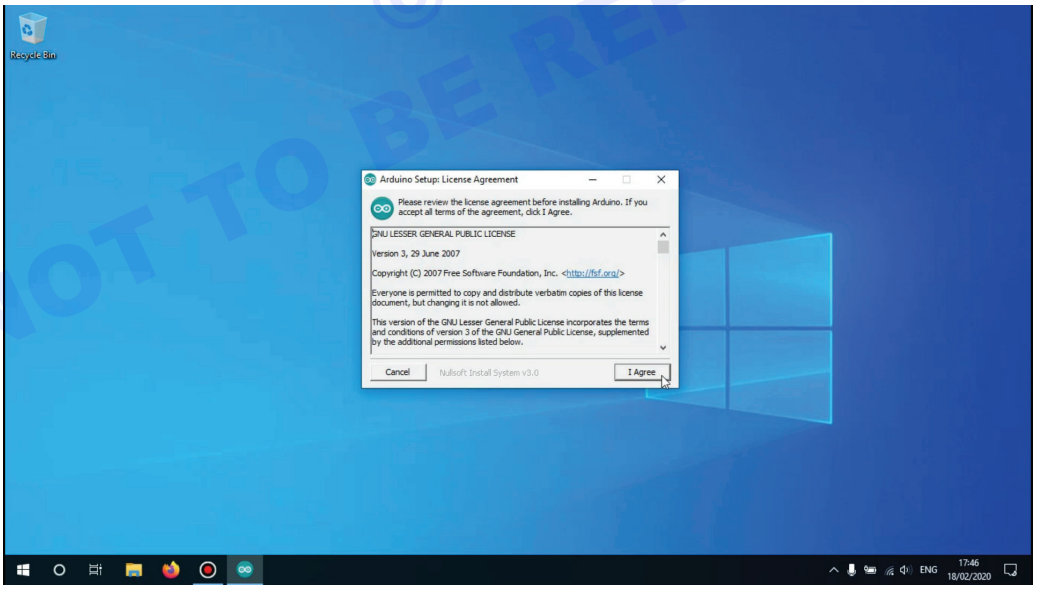
Fig 3



After clicking on the download link you'll be redirected to the donation page, here you can donate or skip it if you like by clicking on the 'Just download' link.

Step 4: Begin the Installation Process (Fig 4)

Fig 4



Open the downloaded file.

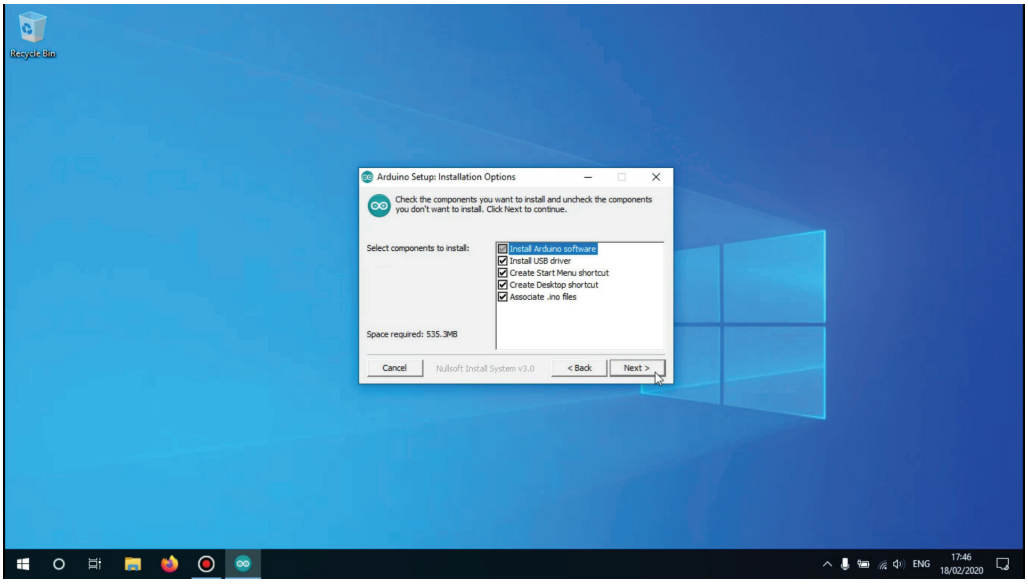
A new window will open asking you to agree to the license agreement.

Click on 'I agree' to continue.

Step 5: Select What to Install (Fig 5)

© NIMI NOT TO BE REPUBLISHED

Fig 5



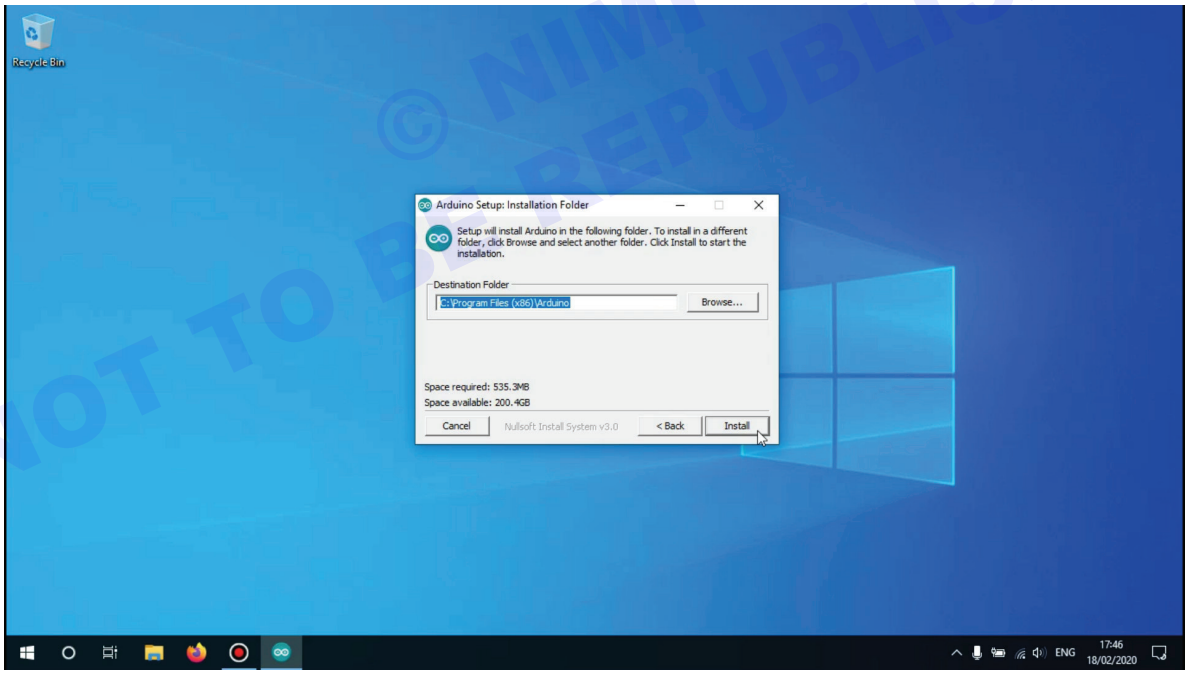
Now you'll see all the available options to install the software with.

If you don't know what you need, it is best to keep everything checked as you can change it later when the installation has finished.

Click on 'Next' to continue.

Step 6: Choose the Installation Path (Fig 6)

Fig 6



Now you have to choose the path the software will be installed in.

it is fine to leave it at the configured location but if you want the Arduino IDE somewhere else installed you can change that here.

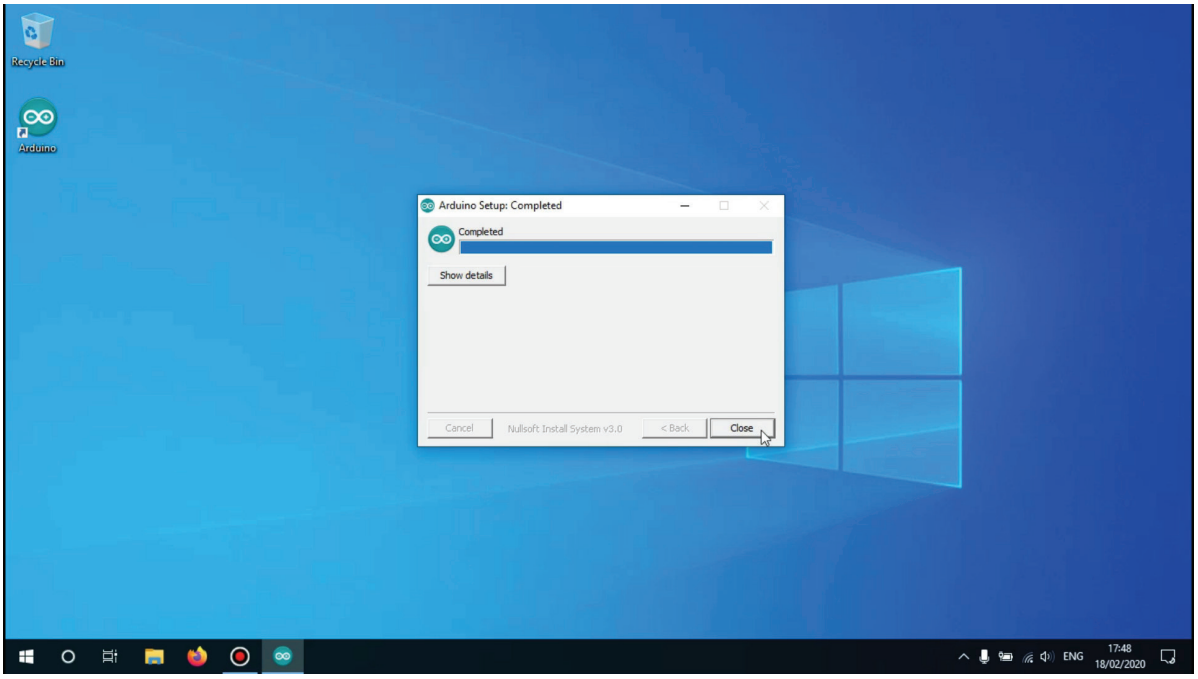
Click on 'Install' to begin the installation

Step 7: Finish the Installation (Fig 7)

Wait until the installation is finished, it shouldn't take very long.

© NIMI NOT TO BE REPUBLISHED

Fig 7



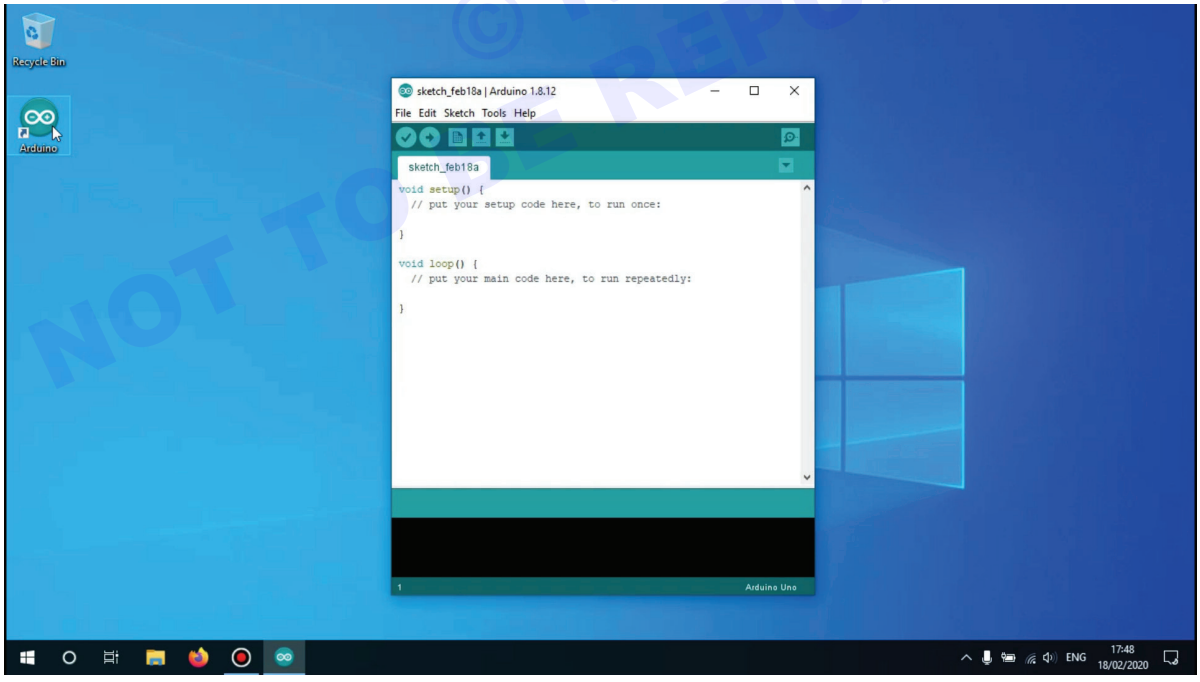
When the installation is finished you may click on 'close' to end the setup wizard.

Step 8 : Launch the Arduino IDE (Fig 8)

The Arduino IDE has now been successfully installed.

To launch the IDE you can click on the Desktop icon that was created for you, or by searching for it in the start menu.

Fig 8



© NIMI NOT TO BE REPUBLISHED

Opening the IDE

Objective : At the end of this exercise you shall be able to
- **launch the IDE to start script development and load testing in loadrunner.**

Requirements	
Tools/Equipments/Machines	
• A working PC with IDE Software	- as reqd.

PROCEDURE

Opening the IDE – Step-by-Step Guide with Output

An Integrated Development Environment (IDE) is used for writing, debugging, and testing code. The process of opening an IDE may vary based on the software you are using (e.g., Eclipse, IntelliJ IDEA, Visual Studio, PyCharm, LoadRunner VuGen IDE).

Below is a general step-by-step guide for opening an IDE with expected output messages.

Step 1 : Locate the IDE on Your System

- Click the Start Menu (Windows) or Launchpad (Mac).
- Search for your IDE name (e.g., Visual Studio Code, Eclipse, VuGen).

Example Output

- "Found Eclipse IDE in the applications list."
- "Found LoadRunner VuGen IDE under Performance Testing Tools."

Step 2 : Launch the IDE

- Click the IDE icon or select Open from the right-click menu.
- If using the command line, type:
  - **Windows:** start <IDE_Name> (e.g., start code for VS Code)
  - **Mac/Linux:** open <IDE_Name> (e.g., open eclipse)

Example Output

- "Eclipse IDE is launching..."
- "LoadRunner VuGen starting..."

Step 3 : Select or Create a Workspace (If Required)

- Some IDEs (like Eclipse, IntelliJ IDEA, and VuGen) require a workspace location.
- Select an existing workspace or create a new one.

Example Output

- "Choose Workspace Directory: /Users/YourName/Projects/MyWorkspace"
- "Workspace loaded successfully!"

Step 4 : Wait for IDE to Load

- The IDE may take a few seconds to load all necessary components.

Example Output

- "Loading plugins and extensions..."
- "IDE started successfully!"

Step 5 : Open a Project or Create a New One

- If an existing project is available:
  - Click File → Open Project and select the folder.
- To create a new project:
  - Click File → New → Project and choose the type (Java, Python, LoadRunner script, etc.).

Example Output

- " Project Ecommerce_Automation loaded successfully!"
- "New LoadRunner Web HTTP/HTML script created."

Step 6: Verify the IDE is Ready

- Check if the code editor, console, and file explorer are displayed.
- Ensure no errors or missing dependencies are shown.

Example Output

- "Welcome to Visual Studio Code – Ready to Code!"
- "VuGen is ready – Start scripting!"

© NIMI NOT TO BE REPUBLISHED

IDE Features

Objective: At the end of this exercise you shall be able to
- **create, edit, debug, and replay test scripts.**

Requirements

Tools/Equipments/Machines
- A working PC with IDE Software - as reqd.

PROCEDURE

IDE Features – Step-by-Step Guide with Output

An Integrated Development Environment (IDE) provides tools for writing, debugging, and managing code efficiently. Below is a step-by-step breakdown of the main IDE features along with expected outputs.

Step 1: Open the IDE
- Launch the IDE (e.g., Visual Studio Code, Eclipse, IntelliJ IDEA, PyCharm, LoadRunner VuGen).
- Select or create a workspace/project.

Example Output
- "IDE started successfully!"
- "Project MyApp loaded."

Step 2: Code Editor
- The central area where you write and edit code.
- Features syntax highlighting, auto-completion, and indentation.

Example Output (Fig 1)

Fig 1

```
print("Hello, World!")
```

"Syntax highlighted and auto-indented correctly!"

Step 3: File Explorer (Project Navigation Panel)
- Displays all project files and folders.
- Helps navigate between different scripts and configuration files.

Example Output
- Project: MyApp
- main.py
- src/
- app.py
- config/
- settings.json
- "File structure loaded successfully!"

Step 4: Toolbar & Menu Bar
- Provides shortcuts for Run, Debug, Save, Open, and Settings.

Example Output
- Click Run
- "Running main.py..."
- Output: "Hello, World!"

Step 5: Debugging Tools
- Allows setting breakpoints, stepping through code, and watching variables.

Example Output
- Breakpoint set at line 2
- Debugging started...
- "Variable message = 'Hello, Debugging!'"
- "Execution paused at breakpoint."

Step 6: Output Console & Terminal
- Displays logs, warnings, and errors during execution.
- Supports command-line commands.

Example Output (Fig 2)

Fig 2

```
Running main.py...
Hello, World!
Process finished with exit code 0.
```

"Program executed without errors!"

Step 7: Extensions & Plugins
- Install additional tools, frameworks, and themes.

Example Output:
- Search "Python Extension"
- "Python extension installed successfully!"

Step 8: Version Control (Git Integration)
- Manage code changes, commits, and branches.

Example Output

"Git repository is up to date!" (Fig 3)

Fig 3

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.
```

© NIMI NOT TO BE REPUBLISHED

Menu bar, toolbar, case pane

Objective: At the end of this exercise you shall be able to
- **access features, quick tools, and script organization.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Menu Bar, Toolbar, and Case Pane – Step-by-Step Guide with Output

Most Integrated Development Environments (IDEs), such as Visual Studio Code, Eclipse, IntelliJ IDEA, LoadRunner VuGen, and PyCharm, contain key UI elements like the Menu Bar, Toolbar, and Case Pane for efficient software development and testing.

Below is a step-by-step breakdown of these elements with expected outputs.

Step 1 : Open the IDE

- Launch the IDE (e.g., Eclipse, Visual Studio Code, IntelliJ IDEA, LoadRunner VuGen).
- Open or create a new project/workspace.

Example Output

- "LoadRunner VuGen IDE started successfully!"
- "Project MyApp opened!"

Step 2 : Explore the Menu Bar

- The Menu Bar is located at the top of the IDE window.
- It contains dropdown menus such as:
  - File – Create, open, save projects
  - Edit – Cut, copy, paste, undo, redo
  - View – Change layout, enable/disable panels
  - Run – Execute scripts and applications
  - Debug – Set breakpoints and troubleshoot
  - Tools – Configure IDE settings and extensions
  - Help – Access documentation and support

Example Output

- Click File → New Project
- "New Project Wizard opened!"
- Click Help → About
- "IDE version: LoadRunner VuGen 2024"

Step 3 : Use the Toolbar

- The Toolbar is a shortcut panel below the Menu Bar.
- It contains icons for quick actions like:
  - New File/Project
  - Save
  - Run
  - Debug
  - Stop Execution
  - Find/Search

Example Output

- Click Run
- "Script executed successfully!"
- Click Debug
- "Execution paused at breakpoint!"

Step 4 : Navigate the Case Pane (VuGen IDE)

- The Case Pane (Script Pane in VuGen) displays:
  - Test cases & actions for LoadRunner VuGen scripts
  - Transactions & checkpoints
  - User actions recorded in GUI-based testing

Example Output

- Case Pane Contents (VuGen):
- Action1 (Login Simulation)
- Action2 (Search Products)
- Action3 (Add to Cart & Checkout)

" Test case structure displayed in Case Pane!"

Step 5: Execute & Monitor

- Combine Menu Bar, Toolbar, and Case Pane for effective test execution.

Example Output

- Click Run(Toolbar)
- Output in Case Pane (Fig 1)

Fig 1

```
Step 1: Launch Application - Passed
Step 2: Login - Passed
Step 3: Add to Cart - Passed
Step 4: Checkout - Failed (Timeout)
```

"Error in Step 4 – Investigate timeout issue!"

© NIMI NOT TO BE REPUBLISHED

Log / reference / UI- element rollup pane

Objective: At the end of this exercise you shall be able to
- **display logs, references, and ui element details.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Log, Reference, and UI-Element Rollup Pane in Selenium IDE – Step-by-Step Guide with Output

Selenium IDE is a Web Automated Testing Tool that allows testers to record, edit, and debug test cases. The Log Pane, Reference Pane, and UI-Element Rollup Pane help in understanding execution details, command references, and UI elements.

Step 1: Open Selenium IDE

1 Open your browser (Chrome/Firefox).

2 Click on the Selenium IDE extension.

3 Create or open an existing project.

Example Output:

- "Selenium IDE launched successfully!"
- "**Project:** Web_Testing_Project loaded!"

Step 2: Understand the Log Pane

- **Purpose:** Displays execution logs, errors, and warnings.
- **Location:** Bottom panel of Selenium IDE.
- **Usage:** Helps in debugging and identifying issues during test execution.

Actions & Expected Output

- Run the test case
- View Log Pane Output (Fig 1)

Fig 1

```
[INFO] Starting test execution...
[INFO] Navigating to https://example.com
[PASS] Clicked on login button.
[FAIL] Unable to locate password field.
[ERROR] Test case failed at step 4.
```

"Test case failed due to missing password field!"

Step 3: Explore the Reference Pane

- **Purpose:** Shows details about the selected Selenium command.
- **Location:** Right-hand side panel in Selenium IDE.
- **Usage:** Helps in understanding command syntax, arguments, and examples.

Actions & Expected Output

- Select command: click
- Reference Pane Output:

"Reference Pane provides details for the selected command!" (Fig 2)

Fig 2

```
Command: click
Target: locator (e.g., id=submit-button)
Value: (optional)
Description: Clicks on an element.
Example: click | id=login-btn |
```

Step 4: Use the UI-Element Rollup Pane

- **Purpose:** Groups multiple UI actions into a single reusable action.
- **Location:** Found in Selenium IDE (Advanced Features).
- **Usage:** Helps in reducing redundancy by combining repeated UI interactions.

Actions & Expected Output

"Create a UI-Element Rollup (Login Rollup)

1 Click "Rollup Pane" → New Rollup.

2 Name it "Login_Action".

3 Add steps

4 Save and execute (Fig 3)

Fig 3

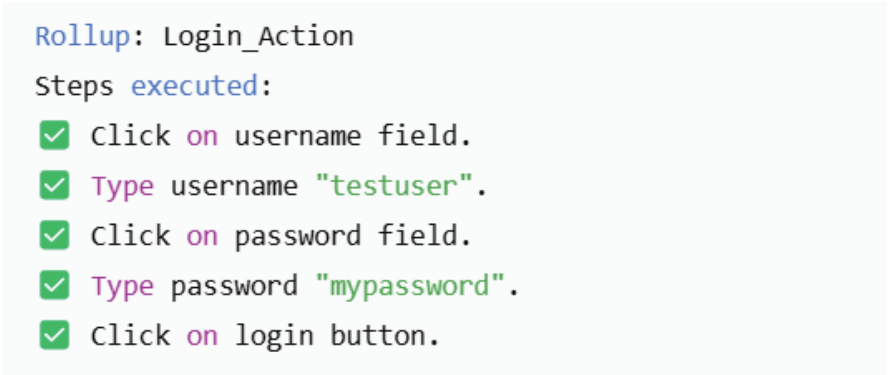
```
click | id=username-field |
type | id=username-field | testuser
click | id=password-field |
type | id=password-field | mypassword
click | id=login-button |
```

© NIMI NOT TO BE REPUBLISHED

- Output in Rollup Pane (Fig 4)

"UI-Element Rollup successfully created and executed!"

Fig 4



© NIMI
NOT TO BE REPUBLISHED

Log, reference, I - element and rollup

Objective: At the end of this exercise you shall be able to
- **display logs, references, ui elements, and summaries.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Log, Reference, UI-Element, and Rollup in Selenium IDE – Step-by-Step Guide with Output

Selenium IDE is a Web Automated Testing Tool that allows testers to record, edit, and debug test cases. Below is a step-by-step guide explaining the Log Pane, Reference Pane, UI-Element Pane, and Rollup Pane in Selenium IDE, along with expected outputs.

Step 1: Open Selenium IDE

1 Open your Chrome or Firefox browser.

2 Click on the Selenium IDE extension.

3 Create a new project or open an existing project.

Example Output

- "Selenium IDE launched successfully!"
- "Project: Web_Testing_Project loaded!"

Step 2: Understanding the Log Pane

Purpose

- Displays real-time execution logs, errors, warnings, and test results.
- Helps in debugging and identifying failures during test execution.

Location

- Found at the bottom panel of Selenium IDE.

Actions & Expected Output

- **Run a test case**
- **View Log Pane Output (Fig 1)**

Fig 1

```
[INFO] Test execution started...
[INFO] Navigating to https://example.com
[PASS] Clicked on "Login" button.
[FAIL] Unable to locate password field.
[ERROR] Test case failed at step 4.
```

"Test case failed due to missing password field!"

Step 3 : Exploring the Reference Pane

Purpose

- Provides details about the selected Selenium command.
- Shows command syntax, required arguments, and examples.

Location

- Found on the right-hand side panel in Selenium IDE.

Actions & Expected Output:

- **Select command: click**
- **Reference Pane Output (Fig 2)**

Fig 2

```
Command: click
Target: locator (e.g., id=submit-button)
Value: (optional)
Description: Clicks on an element.
Example: click | id=login-btn |
```

"Reference Pane provides details for the selected command!"

Step 4: Using the UI-Element Pane

Purpose

- Stores locators of web elements for reuse across test cases.
- Avoids hardcoding locators in every test step.

Location

- Found under Test Suites in Selenium IDE.

Actions & Expected Output

- Add a new UI element:
  1 Click **"UI-Element Pane" → Add New Element.**
  2 Name it **"LoginButton"**.
  3 Set Locator (Fig 3)

© NIMI NOT TO BE REPUBLISHED

Fig 3

```
id=login-button
```

4 Save the element.

- UI-Element Pane Output (Fig 4)

Fig 4

```
Element: LoginButton
Locator: id=login-button
```

"UI element stored successfully!"

Step 5: Creating a Rollup in Selenium IDE

Purpose

- Groups multiple UI actions into a single reusable action.
- Reduces repetitive steps in different test cases.

Location

- Available in Selenium IDE under Advanced Features.

Actions & Expected Output: (Fig 5)

- Create a UI-Element Rollup (Login Rollup)
  1 Click "Rollup Pane" → New Rollup.
  2 Name it "Login_Action".
  3 Add steps:

Fig 5

```
click | id=username-field |
type | id=username-field | testuser
click | id=password-field |
type | id=password-field | mypassword
click | id=login-button |
```

4 Save and execute.

- **Output in Rollup Pane (Fig 6)**

Fig 6

```
Rollup: Login_Action
Steps executed:
✅ Click on username field.
✅ Type username "testuser".
✅ Click on password field.
✅ Type password "mypassword".
✅ Click on login button.
```

"UI-Element Rollup successfully created and executed!"

© NIMI NOT TO BE REPUBLISHED

Building test cases

Objective: At the end of this exercise you shall be able to
- **create and define test scenarios for execution.**

Requirements

Tools/Equipments/Machines
- A working PC with IDE Software - as reqd.

PROCEDURE

Building test cases for your Arduino projects is essential for ensuring that your code works as expected and to catch any potential issues early. Here's a guide on how to approach building test cases for your Arduino projects:

Steps to build test cases for arduino projects

1 Define Requirements
- Clearly outline what your project is supposed to do. Define the expected behavior and functionalities. This will help you identify what needs to be tested.

2 Identify Test Scenarios
- Break down your project into different scenarios that need testing. These can include:
- Input scenarios (e.g., different sensor readings)
- Output scenarios (e.g., LED states, motor movements)
- Edge cases (e.g., maximum and minimum values)
- Failure conditions (e.g., loss of power, sensor failure)

3 Create Test Cases
- For each scenario, write a test case that includes:
- **Test Case ID :** A unique identifier for the test case.
- **Description:** A brief description of what is being tested.
- **Preconditions :** Any conditions that must be met before the test can be executed (e.g., specific hardware setup).
- **Input :** The input values or conditions for the test.
- **Expected Output:** What you expect to happen when the test is run.
- **Actual Output :** What actually happened when the test was executed (filled in after testing).
- **Status :** Pass/Fail based on the comparison of expected and actual output.

Example Test Case

4 Develop Testing Code
- Write code that sets up your testing environment. This code can include:
- Setting up pins for sensors and actuators.
- Reading inputs and comparing them to expected values.
- Using Serial.print() statements to log results for verification.
- If you have multiple test cases, consider creating functions for each test case to keep your code organized.

5 Run Tests
- Upload the test code to your Arduino and run it. Monitor the output in the Serial Monitor.
- Record the actual results and compare them with your expected outputs.

6 Analyze Results
- Review the outcomes of each test case. If a test fails, analyze the reasons:
- Is there an error in your code?
- Are the hardware components functioning as expected?
- Fix any issues identified and re-run the test cases.

7 Document Everything
- Keep a record of all test cases, outcomes, and any changes made based on the testing. This documentation is crucial for future reference and for others who may work on the project.

8 Refine and Repeat
- Testing is an iterative process. As you add new features or make changes, revisit your test cases and add new ones as necessary. (Fig 1)

Fig 1

```
Test Case ID: TC001
Description: Test LED blinking at 1-second interval
Preconditions: Arduino is powered on, LED is connected to pin 13
Input: Blink interval set to 1000 ms
Expected Output: LED should blink on for 1 second and off for 1 second repeatedly.
Actual Output: (To be filled after testing)
Status: (To be filled after testing)
```

© NIMI NOT TO BE REPUBLISHED

Recording

Objective: At the end of this exercise you shall be able to
- **capture user actions to generate test scripts.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Recording in Selenium IDE – Step-by-Step Guide with Output

Selenium IDE is a Web Automated Testing Tool that allows testers to record, edit, and replay test cases without writing code manually. Below is a step-by-step guide to recording test cases in Selenium IDE, along with the expected output.

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.
2 Click on the Selenium IDE extension.
3 Click "Create a New Project" or open an existing project.

Example Output

- "Selenium IDE launched successfully!"
- "New Project: Web_Testing_Project created!"

Step 2: Start Recording

1 Click on "Record a new test".
2 Enter the website URL (e.g., https://example.com).
3 Click "Start Recording".
4 Selenium IDE will launch a new browser window and begin recording actions.

Example Output

"Recording started for URL: https://example.com"

Step 3: Perform Actions on the Web Page

While recording, perform the following actions:

1 Click on the Login button
2 Enter username
3 Enter password
4 Click on Submit button

Recorded Steps (Fig 1)

Fig 1

```
[1] open | https://example.com |
[2] click | id=login-button |
[3] type | id=username-field | testuser
[4] type | id=password-field | mypassword
[5] click | id=submit-button |
```

"All user interactions recorded successfully!"

Step 4: Stop Recording

1 Return to Selenium IDE.
2 Click "Stop Recording".

Example Output

- "Recording stopped!"
- "5 steps recorded in test case 'LoginTest'!"

Step 5 : Save the Test Case

1 Click File → Save Test Case.
2 Enter a test case name, e.g., "Login_Test".
3 Choose a location and save the test.

Example Output

- "Test case 'Login_Test' saved successfully!"

Step 6 : Play back the recorded test case

1 Click the run button (•) in selenium ide.
2 Selenium IDE will execute the recorded steps.

Expected Output in the Log Pane (Fig 2)

Fig 2

```
[INFO] Running test case: Login_Test
[PASS] Navigated to https://example.com
[PASS] Clicked on Login button.
[PASS] Entered username.
[PASS] Entered password.
[PASS] Clicked Submit button.
[PASS] Test case executed successfully!
```

"Test case executed without errors!"

Step 7 : Review execution logs

1 Click on the Log Pane at the bottom of Selenium IDE.
2 View details of each executed step.

Example Output (Fig 3)

Fig 3

```
[INFO] Test execution completed.
[PASS] All steps executed successfully.
```

© NIMI NOT TO BE REPUBLISHED

Adding verifications and asserts with the context menu

Objective: At the end of this exercise you shall be able to
- **insert validation checks and assertions using context menu.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Adding Verifications and Asserts Using the Context Menu in Selenium IDE – Step-by-Step Guide with Output

In Selenium IDE, you can add verifications and asserts to validate web elements and ensure expected behaviors during test execution. This can be done using the context menu (right-click options) while recording or editing a test case.

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Launch Selenium IDE by clicking on the browser extension.

3 Click "Create a New Project" or open an existing project.

Example Output:

- "Selenium IDE launched successfully!"
- "New Project: Web_Testing_Project created!"

Step 2 : Start Recording

1 Click on "Record a new test".

2 Enter the website URL (e.g., https://example.com).

3 Click "Start Recording".

4 Selenium IDE will launch a new browser window and start recording interactions.

Example Output

- **"Recording started for URL:** https://example.com"

Step 3 : Use the Context Menu to Add Verifications

While recording, perform the following steps to add verifications:

1 Right-click on an element you want to verify (e.g., Login Button).

2 In the context menu, select:

- "Verify Text" (to check if the text is present).
- "Verify Element Present" (to check if the element exists on the page).
- "Verify Value" (to check the value of a text field).

Example output in selenium IDE steps pane (Fig 1)

Fig 1

```
[1] open | https://example.com |
[2] verifyElementPresent | id=login-button |
[3] verifyText | id=welcome-message | Welcome, User! |
[4] verifyValue | id=username-field | testuser |
```

"Verifications added successfully!"

Step 4: Use the Context Menu to Add Assertions

Assertions ensure that a test stops immediately if the condition fails.

1 Right-click on an element (e.g., Login Button).

2 In the context menu, select:

- "Assert Text" (ensures the expected text is displayed).
- "Assert Element Present" (ensures the element is on the page).
- "Assert Value" (ensures a field contains the correct value).

© NIMI NOT TO BE REPUBLISHED

Example Output in Selenium IDE Steps Pane (Fig 2)

Fig 2

```
[5] assertElementPresent | id=login-button |
[6] assertText | id=welcome-message | Welcome, User! |
[7] assertValue | id=username-field | testuser |
```

"Assertions added successfully!"

Step 5: Stop Recording

1 Return to Selenium IDE.

2 Click "Stop Recording".

Example Output

- "Recording stopped!"
- "7 steps recorded in test case 'Login_Verification'!"

Step 6 : Run the Test Case

1 Click the Run Button (•) in Selenium IDE.

2 Selenium IDE will execute the test, including the verifications and asserts.

- **Expected Output in the Log Pane (Fig 3)**

Fig 3

```
[INFO] Running test case: Login_Verification
[PASS] Navigated to https://example.com
[PASS] Verified: Element 'login-button' is present.
[PASS] Verified: Text 'Welcome, User!' is displayed.
[PASS] Verified: Username field contains 'testuser'.
[PASS] All assertions passed!
```

"Test case executed successfully with verifications and assertions!"

Step 7: Review Execution Logs

1 Click on the Log Pane at the bottom of Selenium IDE.

2 View details of each executed step.

Example Output (Fig 4)

"Execution logs verified!"

Fig 4

```
[INFO] Test execution completed.
[PASS] All verification and assertion checks passed successfully.
```

© NIMI NOT TO BE REPUBLISHED

Editing, insert command, table view

Objective: At the end of this exercise you shall be able to
- **modify scripts, insert commands, and view data tables.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Editing, Inserting Commands, and Using Table View in Selenium IDE – Step-by-Step Guide with Output

In Selenium IDE, you can edit test cases, insert new commands, and use Table View to manage test steps efficiently. Below is a detailed step-by-step guide with expected output.

Step 1 : Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Open an existing test case or create a new project.

Example Output

- "Selenium IDE launched successfully!"
- "Test case 'LoginTest' loaded!"

Step 2 : Editing a Test Case

1 Select a test case from the left panel.

2 Locate the command you want to edit in the Table View.

3 Click on the command row to modify it.

4 Change the Command, Target, or Value as needed.

Example (Editing a Step)

- Before Editing (Fig 1)

Fig 1

```
[3] type | id=username-field | testuser
```

- After Editing (Fig 2)

Fig 2

```
[3] type | id=username-field | updatedUser
```

"Step updated successfully!"

Step 3: Inserting a New Command

1 Select a step before where you want to insert a new command.

2 Right-click and choose "Insert New Command".

3 Enter the command name (e.g., pause).

4 Enter the target and value if needed.

- Example (Inserting a Pause Command):
- Before (Fig 3)

Fig 3

```
[3] type | id=username-field | testuser
[4] type | id=password-field | mypassword
```

- After (Fig 4)

Fig 4

```
[3] type | id=username-field | testuser
[4] pause | 2000 |
[5] type | id=password-field | mypassword
```

"New command inserted successfully!"

Step 4: Using Table View to Edit Commands

1 Click on Table View (default mode).

2 Modify the command, target, or value directly in the table.

3 Use the drop-down menu in the command column to select commands easily.

© NIMI NOT TO BE REPUBLISHED

Example of Table View (Fig 5)

Fig 5

#	Command	Target	Value
1	open	https://example.com	
2	click	id=login-button	
3	type	id=username-field	testuser
4	pause		2000
5	type	id=password-field	mypassword
6	click	id=submit-button	

"Table View provides an organized way to edit test steps!"

Step 5: Deleting a Command

1 Select the row containing the command.

2 Right-click and choose "Delete".

3 Confirm deletion.

Example (Deleting the Pause Command) (Fig 6)

Fig 6

```
[4] pause | 2000 |  <-- Deleted!
```

"Unnecessary command removed successfully!"

Step 6 : Save and Run the Test

1 Click File → Save Test Case.

2 Click Run (•) to execute the updated test case.

Expected Output in the Log Pane (Fig 7)

Fig 7

```
[INFO] Running test case: LoginTest
[PASS] Navigated to https://example.com
[PASS] Clicked on Login button.
[PASS] Entered updated username.
[PASS] Entered password.
[PASS] Clicked Submit button.
[PASS] Test case executed successfully!
```

"Test executed with updated commands!

© NIMI NOT TO BE REPUBLISHED

Source view, insert comment, table view, source view, edit a command or comment

Objective: At the end of this exercise you shall be able to
- **edit scripts, insert comments, and manage data.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Source View, Insert Comment, Table View, and Editing Commands in Selenium IDE – Step-by-Step Guide with Output

In Selenium IDE, you can view and edit test cases in Table View or Source View, insert comments for better documentation, and modify test steps efficiently. Below is a detailed step-by-step guide with expected output.

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Open an existing test case or create a new project.

Example Output

"Selenium IDE launched successfully!"

"Test case 'LoginTest' loaded!"

Step 2 : Switch to Table View

1 By default, Selenium IDE opens in Table View.

2 If not, click on the "Table View" tab.

3 Table View displays test steps in a structured, easy to-edit format.

Example Table View (Fig 1)

Fig 1

#	Command	Target	Value
1	open	https://example.com	
2	click	id=login-button	
3	type	id=username-field	testuser
4	type	id=password-field	mypassword
5	click	id=submit-button	

© NIMI NOT TO BE REPUBLISHED

"Switched to Table View successfully!"

Step 3 : Switch to Source View

1 Click on the "Source View" tab.

2 Source View shows the test case in JSON format.

Example Source View (JSON format) (Fig 2)

Fig 2

```
{
  "id": "LoginTest",
  "commands": [
    { "command": "open", "target": "https://example.com", "value": "" },
    { "command": "click", "target": "id=login-button", "value": "" },
    { "command": "type", "target": "id=username-field", "value": "testuser" },
    { "command": "type", "target": "id=password-field", "value": "mypassword" },
    { "command": "click", "target": "id=submit-button", "value": "" }
  ]
}
```

"Switched to Source View successfully!"

Step 4: Insert a Comment in Table View

1 Switch back to Table View.

2 Right-click on the row where you want to add a comment.

3 Select "Insert New Command".

4 In the Command column, type comment.

5 In the Target column, enter your comment text (e.g., "Verify login functionality").

Example Table View with Comment (Fig 3)

Fig 3

#	Command	Target	Value
1	open	https://example.com	
2	comment	Verify login functionality	
3	click	id=login-button	
4	type	id=username-field	testuser
5	type	id=password-field	mypassword
6	click	id=submit-button	

"Comment added successfully!"

Step 5: Insert a Comment in Source View

1 Switch to Source View.

2 Locate where you want to add the comment.

3 Add a comment using "comment" as the command.

Example Source View with Comment (Fig 4)

Fig 4

```
{
  "id": "LoginTest",
  "commands": [
    { "command": "open", "target": "https://example.com", "value": "" },
    { "command": "comment", "target": "Verify login functionality", "value": "" },
    { "command": "click", "target": "id=login-button", "value": "" },
    { "command": "type", "target": "id=username-field", "value": "testuser" },
    { "command": "type", "target": "id=password-field", "value": "mypassword" },
    { "command": "click", "target": "id=submit-button", "value": "" }
  ]
}
```

© NIMI NOT TO BE REPUBLISHED

"Comment added successfully in Source View!"

Step 6: Edit a Command or Comment in Table View

1 Click on the command row you want to modify.

2 Update the Command, Target, or Value as needed.

3 Press Enter to save changes.

Example (Editing Username Field Entry)

- Before Editing (Fig 5)

Fig 5

```
[4] type | id=username-field | testuser
```

- After Editing (Fig 6)

Fig 6

```
[4] type | id=username-field | updatedUser
```

"Command edited successfully!"

Step 7: Edit a Command or Comment in Source View

1 Switch to Source View.

2 Modify the command or comment text directly in the JSON structure.

3 Ensure proper JSON syntax and save changes.

Example (Editing a Comment in Source View)

- Before Editing (Fig 7)

Fig 7

```
{ "command": "comment", "target": "Verify login functionality", "value": "" }
```

- After Editing (Fig 8)

Fig 8

```
{ "command": "comment", "target": "Ensure login button is visible before clicking", "value": "" }
```

"Comment edited successfully in Source View!"

Step 8: Save and Run the Test

1 Click File → Save Test Case.

2 Click Run (•) to execute the updated test case.

Expected Output in the Log Pane (Fig 9)

"Test executed with updated commands and comments!"

Fig 9

```
[INFO] Running test case: LoginTest
[PASS] Navigated to https://example.com
[INFO] Verify login functionality
[PASS] Clicked on Login button.
[PASS] Entered updated username.
[PASS] Entered password.
[PASS] Clicked Submit button.
[PASS] Test case executed successfully!
```

© NIMI NOT TO BE REPUBLISHED

Table view, source view, opening and saving a test case

Objective: At the end of this exercise you shall be able to
- **view, edit, and manage test cases efficiently.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Table View, Source View, Opening, and Saving a Test Case in Selenium IDE – Step-by-Step Guide with Output

Selenium IDE provides two ways to view test cases: Table View and Source View. You can also open and save test cases for future execution. This guide explains these features step by step with expected outputs.

Step 1 : Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Start a new project or open an existing test case.

Example Output

"Selenium IDE launched successfully!"

"Test case 'LoginTest' loaded!"

Step 2 : Open a Test Case

1 Click File → Open.

2 Navigate to the saved test case file (.side file).

3 Select the file and click Open.

Expected Output

"Test case 'LoginTest.side' opened successfully!"

Step 3 : Switch to Table View

1. If Table View is not visible, click on the Table View tab.
2. In this mode, test steps are displayed in a structured format with Command, Target, and Value fields.

Example Table View (Fig 1)

Fig 1

#	Command	Target	Value
1	open	https://example.com	
2	click	id=login-button	
3	type	id=username-field	testuser
4	type	id=password-field	mypassword
5	click	id=submit-button	

"Switched to Table View successfully!"

Step 4 : Switch to Source View

1 Click on the Source View tab.

2 Source View shows the test case in JSON format.

Example Source View (JSON format) (Fig 2)

Fig 2

```
{
  "id": "LoginTest",
  "commands": [
    { "command": "open", "target": "https://example.com", "value": "" },
    { "command": "click", "target": "id=login-button", "value": "" },
    { "command": "type", "target": "id=username-field", "value": "testuser" },
    { "command": "type", "target": "id=password-field", "value": "mypassword" },
    { "command": "click", "target": "id=submit-button", "value": "" }
  ]
}
```

© NIMI NOT TO BE REPUBLISHED

"Switched to Source View successfully!"

Step 5: Edit the Test Case in Table View

1 Click on a row in Table View to modify it.

2 Change the Command, Target, or Value as needed.

3 Press Enter to save changes.

Example (Editing a Step in Table View)

- Before Editing (Fig 3)

Fig 3

```
[3] type | id=username-field | testuser
```

- After Editing (Fig 4)

Fig 4

```
[3] type | id=username-field | updatedUser
```

"Test step updated successfully!"

Step 6: Save the Test Case

1 Click File → Save Test Case As.

2 Choose a location and enter a filename (e.g., LoginTest.side).

3 Click Save.

Expected Output

"Test case 'LoginTest.side' saved successfully!"

Step 7: Run the Test Case

1 Click Run (•) to execute the test.

2 Selenium IDE will automate browser actions.

Expected Log Output (Fig 5)

Fig 5

```
[INFO] Running test case: LoginTest
[PASS] Navigated to https://example.com
[PASS] Clicked on Login button.
[PASS] Entered updated username.
[PASS] Entered password.
[PASS] Clicked Submit button.
[PASS] Test case executed successfully!
```

"Test executed successfully!"

© NIMI NOT TO BE REPUBLISHED

Running test cases

Objective: At the end of this exercise you shall be able to
- **execute test cases and monitor performance results.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Running Test Cases in Selenium IDE – Step-by-Step Guide with Output

Selenium IDE allows you to record, edit, and run test cases in a simple and efficient manner. Below is a step-by-step guide on how to execute test cases and analyze the output.

Step 1 : Open Selenium IDE

1 Open google chrome or mozilla firefox.

2 Click on the Selenium IDE extension.

3 Start a new project or open an existing test case.

Expected Output

- "Selenium IDE launched successfully!"
- "Test case 'LoginTest' loaded!"

Step 2 : Open the Test Case to Run

1 Click File → Open.

2 Select the.side file containing your test case.

3 Click Open.

Expected Output

- "Test case 'LoginTest.side' opened successfully!"

Step 3 : Verify the test steps in table view

1 Ensure that the test steps are correct.

2 Modify any steps if needed before execution.

Example Test Case (Table View) (Fig 1)

Fig 1

#	Command	Target	Value
1	open	https://example.com	
2	click	id=login-button	
3	type	id=username-field	testuser
4	type	id=password-field	mypassword
5	click	id=submit-button	

"Test case is ready for execution!"

Step 4: Run the Test Case

1 Click the Run Current Test (•) button to execute a single test case.

2 Alternatively, click Run All (•) to execute all test cases in the project.

Expected output in the log pane (Fig 2)

Fig 2

```
[INFO] Running test case: LoginTest
[PASS] Navigated to https://example.com
[PASS] Clicked on Login button.
[PASS] Entered username: testuser
[PASS] Entered password: mypassword
[PASS] Clicked Submit button.
[PASS] Test case executed successfully!
```

"Test case executed successfully!

Step 5: Analyze the Execution Results

1 Check the Log Pane to verify passed and failed steps.

2 Look for errors in the Reference Pane if any step failed.

Example Output (If a Step Fails) (Fig 3)

Fig 3

```
[INFO] Running test case: LoginTest
[PASS] Navigated to https://example.com
[PASS] Clicked on Login button.
[FAIL] Unable to find element: id=username-field
[INFO] Stopping execution due to error.
```

© NIMI NOT TO BE REPUBLISHED

- "Test case failed: Unable to locate element 'id=username-field'"
- "Use the 'Find' button in Selenium IDE to debug missing elements!"

Step 6: Debugging a Failed Test Case (Optional)

1 Use Breakpoints – Right-click on a step and select "Toggle Breakpoint".

2 Run Step-by-Step – Click Run Current Step (•) to execute one step at a time.

3 Modify the Test Steps – Update the Target value if the element locator has changed.

4 Re-run the test after making necessary fixes.

"Debugging helps identify and resolve test failures!"

Step 7: Save the Execution Results

1. Click File → Export Logs to save test execution logs.
2. Choose a format (.log, .txt, .json) and click Save.

"Test execution log saved successfully!"

© NIMI
NOT TO BE REPUBLISHED

Using base URL to run test cases in different domains

Objective: At the end of this exercise you shall be able to
- **set base URL to run tests across multiple domains.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Using Base URL to Run Test Cases in Different Domains in Selenium IDE – Step-by-Step Guide with Output

The Base URL feature in Selenium IDE allows test cases to be executed across different domains by modifying the base URL dynamically instead of hardcoding it in test steps. This makes test cases more reusable and flexible.

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Create a new project or open an existing test case.

Expected Output

- "Selenium IDE launched successfully!"
- "Project 'MyTests' opened!"

Step 2: Set the Base URL

1 Locate the Base URL field at the top of Selenium IDE.

2 Enter the main domain where the test case should start.
   - Example: https://example.com

Expected Output

- "Base URL set to 'https://example.com'"

Step 3: Write a Test Case Without Hardcoded URLs

1 Go to the Table View tab.

2 Ensure test steps use relative paths instead of full URLs.

3 Example test case. (Fig 1)

Fig 1

#	Command	Target	Value
1	open	/login	
2	type	id=username-field	testuser
3	type	id=password-field	mypassword
4	click	id=submit-button	

"Test case created using relative paths!"

Step 4: Run the Test Case on the Default Base URL

1 Click the Run Current Test (•) button.

2 Selenium IDE will prepend the Base URL to relative paths.

3 The browser will navigate to https://example.com/login.

Expected Log Output (Fig 2)

"Test case executed successfully using the default Base URL!"

Fig 2

```
[INFO] Running test case: LoginTest
[PASS] Navigated to https://example.com/login
[PASS] Entered username: testuser
[PASS] Entered password: mypassword
[PASS] Clicked Submit button.
[PASS] Test case executed successfully!
```

Step 5: Change the Base URL to a Different Domain

1 Modify the Base URL field to another domain, e.g., https://test.example.com.

2 Click Run Current Test (•) again.

© NIMI NOT TO BE REPUBLISHED

Expected Output

- "Base URL changed to 'https://test.example.com'"
- "Test case now running on 'https://test.example.com/login'"

Updated Log Output (Fig 3)

Fig 3

```
[INFO] Running test case: LoginTest
[PASS] Navigated to https://test.example.com/login
[PASS] Entered username: testuser
[PASS] Entered password: mypassword
[PASS] Clicked Submit button.
[PASS] Test case executed successfully!
```

"Test case executed successfully on a different domain!"

Step 6: Save the Test Case

1 Click File → Save Test Case As.

2 Choose a filename (e.g., LoginTest.side) and save it.

"Test case saved successfully!"

© NIMI NOT TO BE REPUBLISHED

Selenium commands -"Selenese"

Objective: At the end of this exercise you shall be able to
- **automated commands for web interactions and testing.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

In Selenium IDE, commands written to interact with a web page are called Selenese commands. These commands instruct Selenium on what actions to perform in a test case, such as clicking a button, typing text, verifying conditions, and much more. Selenese commands are broadly categorized based on their functionality.

Here's a breakdown of Selenese commands and how they're commonly used:

1 Action Commands

- Action commands perform actions directly on web elements (e.g., buttons, links, text fields).

Examples

- **click :** Clicks an element, like a button or link.
- **type :** Enters text into an input field.
- **select :** Chooses an option from a dropdown list.
- **open :** Navigates to a specific URL or relative path.

Usage Example (Fig 1)

Fig 1

```
click | id=login-button
type | id=username | testUser
select | id=country-dropdown | label=United States
```

2 Assertion Commands

- Assertion commands validate that an element or condition is present or meets expected criteria. If an assertion fails, the test stops.

Examples

- **Assert text:** Asserts that a specific text is present within an element.
- **Assert title:** Verifies that the page title matches the expected title.
- **Assert element present:** Asserts that a specified element is present on the page.

Usage Example (Fig 2)

Fig 2

```
assertText | id=welcome-message | Welcome, User!
assertTitle | Dashboard
assertElementPresent | id=logout-button
```

3 Verification Commands

- Verification commands are similar to assertions but don't stop the test if they fail. Instead, they log the failure and continue.

Examples

- **Verify text:** Verifies that specific text appears within an element.
- **Verify element present:** Checks that an element is present without halting the test if it's missing.
- **Verify title:** Checks that the page title matches the expected title.

Usage Example (Fig 3)

Fig 3

```
verifyText | id=user-name | John Doe
verifyTitle | My Profile
verifyElementPresent | id=profile-picture
```

4 Wait Commands

- Wait commands pause the test execution until a specified condition is met, often used to ensure elements are loaded before interacting with them.

Examples

- waitForElementPresent: Waits until a specific element is present on the page.
- waitForText: Waits until specific text appears within an element.
- waitForTitle: Pauses until the page title matches the expected title.

© NIMI NOT TO BE REPUBLISHED

Usage Example (Fig 4)

Fig 4

```
waitForElementPresent | id=search-bar
waitForText | id=notification | Welcome back!
waitForTitle | Home Page
```

5 Store Commands

- Store commands are used to save data (such as text values or attributes) in variables, allowing you to reuse that data in later commands.

Examples

- **Store text:** Saves text from an element to a variable.
- **Store value:** Captures a value (such as an input field value) and stores it.
- **Store title:** Stores the page title in a variable.

Usage Example (Fig 5)

Fig 5

```
storeText | id=user-greeting | greetingMessage
storeValue | id=input-email | userEmail
storeTitle | pageTitle
```

6 Other Utility Commands

- These commands serve various purposes, such as executing JavaScript or handling custom scenarios.

Examples

- **Echo:** Outputs a message to the log, useful for debugging.
- **Run script:** Executes custom JavaScript code on the page.
- **Pause:** Temporarily halts the test for a set period.

Usage Example (Fig 6)

Fig 6

```
echo | Starting login test
runScript | return document.title;
pause | 3000
```

7 Flow Control Commands (in Advanced Selenium IDE Versions)

- Some advanced versions of Selenium IDE support flow control commands like if statements, for Each loops, and while loops, allowing you to create more complex test flows.

Example of an if condition (Fig 7)

Fig 7

```
if | ${greetingMessage} == Welcome back!
  echo | User is recognized
end
```

© NIMI NOT TO BE REPUBLISHED

Script syntax

Objective: At the end of this exercise you shall be able to
- **define rules for writing and executing test scripts.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

In Selenium IDE, script syntax relies on Selenese commands structured in a three-column format of Command, Target, and Value. Here's an overview of the syntax and how it works:

Basic Script Structure

Each line in a Selenium IDE script follows this structure. (Fig 1)

Fig 1

```
Command | Target | Value
```

- **Command:** The action to perform (e.g., click, type, verifyText).
- **Target:** The element locator (e.g., id=login-button), specifying where the command will act.
- **Value:** The input or parameter for the command (e.g., text to enter in a field), if required.

Example Script Syntax

Here's a script example that performs a login on a web page. (Fig 2)

Fig 2

```
open | /login |
type | id=username | testUser
type | id=password | password123
click | id=login-button |
assertText | id=welcome-message | Welcome, testUser!
```

Locating Elements with Target Syntax

Target defines the element locator, which identifies where the action should be performed on the page. Selenium supports multiple locator strategies:

1 **ID :** id=element-id

2 **Class Name :** class=class-name

3 **Name Attribute :** name=element-name

4 **Link Text (for hyperlinks) :** link=Link Text

5 **CSS Selector :** css=css-selector

6 **XPath :** xpath=xpath-expression

Common Command Syntax Examples

Below are common Selenese commands with example syntax.

1 Navigation and Actions

- **Open:** Opens a specified URL or relative path. (Fig 3)

Fig 3

```
open | /home |
```

- **Click:** Clicks on an element. (Fig 4)

Fig 4

```
click | id=submit-button |
```

- **Type:** Enters text into an input field. (Fig 5)

Fig 5

```
type | id=username | testUser
```

- **Select:** Selects an option from a dropdown menu. (Fig 6)

Fig 6

```
select | id=country-dropdown | label=United States
```

2 Assertions and Verifications

- **assertText:** Confirms that an element contains the specified text. (Fig 7)

Fig 7

```
assertText | id=welcome | Welcome, User!
```

- **verifyElementPresent:** Checks if an element exists. (Fig 8)

Fig 8

```
verifyElementPresent | id=logout-button |
```

© NIMI NOT TO BE REPUBLISHED

- **assertTitle:** Verifies that the page title matches a specified value. (Fig 9)

Fig 9

```
assertTitle | Home Page
```

3 Wait Commands

- **waitForElementPresent:** Waits until an element appears on the page. (Fig 10)

Fig 10

```
waitForElementPresent | id=notification |
```

- **waitForText:** Pauses until a specified text appears within an element. (Fig 11)

Fig 11

```
waitForText | id=status-message | Logged in successfully
```

4 Store Commands (Using Variables)

- **storeText:** Saves text from an element into a variable. (Fig 12)

Fig 12

```
storeText | id=welcome-message | userWelcomeMessage
```

- **storeValue:** Captures the value from a field and stores it. (Fig 13)

Fig 13

```
storeValue | id=username | storedUsername
```

Example of Using Variables in Script Syntax

Variables can be referenced using the ${variableName} format. (Fig 14)

Fig 14

```
open | /profile |
storeText | id=user-name | userName
assertText | id=user-greeting | Welcome, ${userName}!
```

Comments

Comments can be added to your Selenium IDE script to improve readability. (Fig 15)

Fig 15

```
// This is a comment line for documentation
click | id=login-button |
```

Example of a Full Login Test Script

Here's a complete login and verification script, demonstrating various commands. (Fig 16)

Fig 16

```
open | /login |
type | id=username | testUser
type | id=password | password123
click | id=login-button |
waitForElementPresent | id=welcome-message |
assertText | id=welcome-message | Welcome, testUser!
```

Conditional Logic and Control Flow (Advanced)

In newer Selenium IDE versions, if statements, while loops, and for Each loops allow more complex scripts. (Fig 17)

Fig 17

```
if | ${userName} == Admin
    click | id=admin-settings |
end

while | ${retryCount} < 3
    click | id=retry-login |
    storeEval | ${retryCount}+1 | retryCount
end
```

© NIMI NOT TO BE REPUBLISHED

Test suites

Objective: At the end of this exercise you shall be able to
- **group related test cases for efficient execution.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Test suites in selenium IDE – Step-by-step guide with output

A test suite in selenium ide is a collection of test cases that run sequentially. instead of running test cases one by one, you can group multiple tests together and execute them as a suite. this helps in batch execution and regression testing.

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Start a new project or open an existing project.

Expected Output:

"Selenium IDE launched successfully!"

Project 'MyTests' opened!"

Step 2 : Create a New Test Suite

1 Click "File" → "New Test Suite".

2 Enter a name for the suite (e.g., LoginSuite).

3 Click "OK".

Expected Output

"Test suite 'LoginSuite' created successfully!"

Step 3: Add Test Cases to the Test Suite

1 Click "Add Test Case" in the Test Suite panel.

2 Select test cases you want to include.
- Example: LoginTest, LogoutTest, SignupTest.

3 Click "Add".

Expected Output

"Test cases added to 'LoginSuite'!"

Example test suite structure (Fig 1)

Fig 1

```
LoginSuite
├── LoginTest
├── LogoutTest
└── SignupTest
```

Step 4: Run the Entire Test Suite

1 Click Run Test Suite (•) to execute all test cases in sequence.

2 Selenium IDE will run the tests one by one in the browser.

Expected Log Output

"Test suite executed successfully!" (Fig 2)

Fig 2

```
[INFO] Running test suite: LoginSuite
[INFO] Running test case: LoginTest
[PASS] Navigated to https://example.com/login
[PASS] Entered username and password
[PASS] Clicked Submit button.
[INFO] Running test case: LogoutTest
[PASS] Clicked Logout button.
[INFO] Running test case: SignupTest
[PASS] Entered registration details.
[PASS] Clicked Signup button.
[PASS] Test suite executed successfully!
```

© NIMI NOT TO BE REPUBLISHED

Step 5: Analyze the Execution Results

1 Check the Log Pane to see which tests passed or failed.

2 Look for error messages in case of failures.

Example Log Output (if a test fails): (Fig 3)

"Test suite failed: LoginTest encountered an issue!" (Fig 3)

Fig 3

```
[INFO] Running test case: LoginTest
[FAIL] Unable to find element: id=username-field
[INFO] Stopping execution due to error.
```

"Use debugging tools to fix the issue before re-running the suite!"

Step 6: Save the Test Suite

1 Click File → Save Test Suite As.

2 Choose a filename (e.g., LoginSuite.side) and click Save.

Expected Output

"Test suite saved successfully!"

© NIMI NOT TO BE REPUBLISHED

Commonly used selenium commands

Objective: At the end of this exercise you shall be able to
- **automate web actions with essential selenium commands.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Commonly Used Selenium Commands in Selenium IDE– Step-by-Step Guide with Output

Selenium IDE provides a range of commands (selenese) to interact with web applications. these commands help automate navigation, user actions, and validation.

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Start a new project or open an existing test case.

Expected Output

"Selenium IDE launched successfully!"

"Test case ready for editing!"

Step 2: Use Commonly Used Commands

1 Navigation Commands (Fig 1)

Fig 1

Command	Description	Example	Output
`open`	Opens a URL	`open	https://example.com
`goBack`	Navigates back in browser history	`goBack	
`refresh`	Refreshes the page	`refresh	

2 Interaction Commands (Fig 2)

Fig 2

Command	Description	Example	Output
`click`	Clicks an element	`click	id=submit-button
`type`	Enters text in an input field	`type	id=username
`sendKeys`	Sends keystrokes (e.g., Enter, Tab)	`sendKeys	id=search

3 Verification Commands (Fig 3)

Fig 3

Command	Description	Example	Output
`verifyText`	Checks if a text is present	`verifyText	id=welcome
`assertTitle`	Confirms page title	`assertTitle	Home Page
`verifyElementPresent`	Checks if an element exists	`verifyElementPresent	id=login-button

4 Wait Commands (Fig 4)

Fig 4

Command	Description	Example	Output
`waitForElementPresent`	Waits for an element to appear	`waitForElementPresent	id=loader
`waitForText`	Waits until text appears	`waitForText	id=message

© NIMI NOT TO BE REPUBLISHED

5 Store and Print Variables (Fig 5)

Fig 5

Command	Description	Example	Output
`store`	Stores a value in a variable	`store	testuser
`echo`	Prints text in the log	`echo	Test Passed

Step 3: Run the test case

1 Click the run current test (•) button.

2 Selenium IDE will execute all commands in sequence.

Expected log output

"Test case executed successfully!" (Fig 6)

Fig 6

```
[INFO] Running test case: LoginTest
[PASS] Navigated to https://example.com
[PASS] Typed 'testuser' in username field
[PASS] Clicked Submit button.
[PASS] Page title matches: Home Page
[PASS] Test case executed successfully!
```

© NIMI NOT TO BE REPUBLISHED

Verifying page elements

Objective: At the end of this exercise you shall be able to
- **check if web elements exist and function correctly.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE Software - as reqd.

PROCEDURE

Verifying page elements is a crucial part of automated testing with Selenium, ensuring that the web application behaves as expected. Here are common methods and techniques for verifying various elements on a webpage using Selenium, along with example code snippets.

Common verification methods

1 Check Element Presence

- Verify if an element is present on the page. (Fig 1)

Fig 1

```
from selenium import webdriver
from selenium.common.exceptions import NoSuchElementException

driver = webdriver.Chrome()
driver.get("http://example.com")

try:
    element = driver.find_element_by_id("elementId")
    print("Element is present")
except NoSuchElementException:
    print("Element is not present")
```

© NIMI NOT TO BE REPUBLISHED

2 Verify element visibility

- Ensure that an element is visible (i.e., not hidden). (Fig 2)

Fig 2

```
from selenium.webdriver.common.by import By

element = driver.find_element(By.ID, "elementId")
if element.is_displayed():
    print("Element is visible")
else:
    print("Element is not visible")
```

3 Verify element text

- Check if the text of an element matches the expected value. (Fig 3)

Fig 3

```
expected_text = "Expected Text"
actual_text = driver.find_element(By.ID, "elementId").text
assert actual_text == expected_text, f"Expected '{expected_text}', but got '{actual_text}'"
```

4 Verify element attribute

- Check the value of a specific attribute of an element. (Fig 4)

Fig 4

```
expected_value = "expectedValue"
actual_value = driver.find_element(By.ID, "elementId").get_attribute("attributeName")
assert actual_value == expected_value, f"Expected '{expected_value}', but got '{actual_value
```

5 Verify checkbox/radio button status

- Ensure that a checkbox is checked or unchecked. (Fig 5)

Fig 5

```
checkbox = driver.find_element(By.ID, "checkboxId")
assert checkbox.is_selected(), "Checkbox is not checked"
```

6 Verify Page Title

- Check if the page title matches the expected title. (Fig 6)

Fig 6

```
expected_title = "Expected Page Title"
assert driver.title == expected_title, f"Expected title '{expected_title}', but got '{driver
```

7 Verify URL

- Check if the current URL is as expected. (Fig 7)

Fig 7

```
expected_url = "http://example.com/home"
assert driver.current_url == expected_url, f"Expected URL '{expected_url}', but got '{driver
```

© NIMI NOT TO BE REPUBLISHED

Assertion or Verification?

Objective: At the end of this exercise you shall be able to
- **validate expected outcomes during test execution.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

Assertion vs. Verification in Selenium IDE – Step-by-Step Guide with Output

In Selenium IDE, both Assertions and Verifications are used to validate test conditions, but they behave differently when a test fails:

Assertions: Stops test execution if the condition fails.

Verifications: Continues execution even if the condition fails.

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Start a new project or open an existing test case.

Expected Output:

"Selenium IDE launched successfully!"

"Test case ready for execution!"

Step 2: Using Assertions in Selenium IDE

Assertions are strict checks. If they fail, the test stops immediately (Fig 1)

Fig 1

Command	Target	Value	Expected Output
`assertText`	id=message	Welcome User	✅ Test continues if text is present; ❌ Test fails and stops otherwise
`assertTitle`	Home Page		✅ Pass if title matches; ❌ Fail and stop otherwise
`assertElementPresent`	id=login-button		✅ Pass if element exists; ❌ Fail and stop otherwise

Example test case using assertions (Fig 2)

Fig 2

#	Command	Target	Value
1	open	https://example.com	
2	assertTitle	Home Page	
3	assertText	id=message	Welcome User

Expected Log Output

If all assertions pass (Fig 3)

Fig 3

```
[PASS] Page title matches: Home Page
[PASS] Found text: Welcome User
[Test case executed successfully!]
```

© NIMI NOT TO BE REPUBLISHED

If an assertion fails (e.g., wrong title) (Fig 4)

Fig 4

```
[FAIL] Expected title 'Home Page' but found 'Dashboard'
[Test execution stopped!]
```

Key Takeaway: Assertions stop the test immediately if a condition fails.

Step 3: Using Verifications in Selenium IDE

Verifications log failures but allow the test to continue. (Fig 5)

Fig 5

Command	Target	Value	Expected Output
`verifyText`	id=message	Welcome User	✅ Logs failure but continues execution
`verifyTitle`	Home Page		✅ Logs failure but continues execution
`verifyElementPresent`	id=login-button		✅ Logs failure but continues execution

Example test case using verifications. (Fig 6)

Fig 6

#	Command	Target	Value
1	open	https://example.com	
2	verifyTitle	Home Page	
3	verifyText	id=message	Welcome User
4	click	id=login-button	

© NIMI NOT TO BE REPUBLISHED

Expected Log Output

If all verifications pass (Fig 7)

Fig 7

```
[PASS] Page title matches: Home Page
[PASS] Found text: Welcome User
[PASS] Clicked Login button
[Test case executed successfully!]
```

If a verification fails (e.g., wrong title), the test still runs. (Fig 8)

Fig 8

```
[FAIL] Expected title 'Home Page' but found 'Dashboard'
[INFO] Continuing test execution...
[PASS] Found text: Welcome User
[PASS] Clicked Login button
[Test case executed with warnings!]
```

Key takeaway: Verifications allow the test to continue execution even if a condition fails.

Verify text present, verify element present

Objective: At the end of this exercise you shall be able to
- **check if text or element exists on page.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

In Selenium IDE, a browser-based web testing tool, you can use the following commands to perform "verify text present" and "verify element present" checks to validate that specific content or elements are present on a web page. Here's how each is implemented :

1 Verify Text Present

- In selenium IDE, to verify that specific text is present on a page, use the command verify text or verify text present.
- **Command:** verifyText
- **Target:** The CSS/XPath locator of the element containing the text.
- **Value:** The exact text you expect to be present.

Example (Fig 1)

Fig 1

```
Command: verifyText
Target: css=div.alert
Value: Submission successful
```

- This will check if the text "Submission successful" appears within the element located by css=div.alert. If the text is not present, the test will record a failure but continue running (as verify commands are soft assertions in Selenium IDE).

2 Verify Element Present

- To verify that a specific element is present on the page, use verifyElementPresent.
- **Command:** verifyElementPresent
- **Target:** The CSS/XPath locator of the element you want to check.

Example (Fig 2)

Fig 2

```
Command: verifyElementPresent
Target: css=button#save
```

- This checks if the element with button#save is present in the DOM. If the element isn't found, Selenium IDE will record a failure but proceed with the test execution.

Key Points

- **Soft assertions:** verify commands don't stop the test if they fail; they log an error and continue, making them suitable for checking multiple elements without interrupting the test flow.
- **Locators: U**se precise locators (CSS or XPath) to ensure you're targeting the right elements.

© NIMI NOT TO BE REPUBLISHED

Verify text

Objective: At the end of this exercise you shall be able to
- **ensure specified text is present on the page.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

Verify Text in Selenium IDE – Step-by-Step Guide with Output (Fig 1)

In Selenium IDE, the verifyText command is used to check whether a specific text appears on the webpage. Unlike assertText, verification does not stop the test execution if the condition fails—it simply logs the result.

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Start a new project or open an existing test case.

Expected Output :

"Selenium IDE launched successfully!"

"Test case ready for editing!"

Step 2: Add the verifyText Command

1 Open Selenium IDE.

2 Click on Table View.

3 Add the following command (Fig 1)

Fig 1

Command	Target	Value
open	https://example.com	
verifyText	id=welcome-message	Welcome User

Explanation

- Open → Navigates to the website.
- verifyText → Checks if the element with ID welcome-message contains "Welcome User".

Step 3: Run the Test Case

1 Click the Run Current Test Case (•) button.

2 Selenium IDE will check if the text "Welcome User" is present.

Expected Log Output (When text is found) (Fig 2)

Fig 2

```
[PASS] Found text: "Welcome User" in element id=welcome-message
[Test case executed successfully!]
```

If the text is NOT found (Fig 3)

Fig 3

```
[FAIL] Expected text "Welcome User" but found "Hello Guest"
[INFO] Test case continued execution despite failure.
```

Test execution continues even if verification fails!

Step 4: Using verifyTextPresent (Alternative Approach)

Instead of verifying text inside a specific element, you can check if the text exists anywhere on the page using verifyTextPresent. (Fig 4)

Fig 4

Command	Target	Value
open	https://example.com	
verifyTextPresent	Welcome User	

Expected Log Output

If text is found (Fig 5)

Fig 5

```
[PASS] "Welcome User" is present on the page
```

If text is NOT found (Fig 6)

Fig 6

```
[FAIL] Expected text "Welcome User" but not found on the page
```

© NIMI NOT TO BE REPUBLISHED

Location elements

Objective: At the end of this exercise you shall be able to
- **identify and target web elements for interaction.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

In Selenium IDE and other web testing tools, locating elements is essential to identify specific items on a web page, such as buttons, text fields, images, or links. Selenium IDE uses various locators to pinpoint these elements precisely. The most common locators in Selenium are:

1 ID Locator

- **Syntax :** id=element_id
- **Example :** id=loginButton
- This locator targets an element with a unique id attribute. It's one of the fastest locators as IDs are typically unique to each element on the page.

2 Name Locator

- **Syntax :** name=element_name
- **Example :** name=username
- Targets an element with a name attribute. Often used for form fields like inputs and buttons. The name attribute is usually unique but not always, so use carefully.

3 CSS Selector

- **Syntax :** css=css_selector
- **Example :** css=button.submit-btn or css=div#main .header h1
- CSS selectors can be used to locate elements based on classes, IDs, attributes, and more. They are powerful and can be very specific.

4 XPath

- **Syntax :** xpath=xpath_expression
- **Example :** xpath=//button[@id='submit']
- XPath is highly flexible and can navigate through the HTML structure. It's helpful for complex locators but can be slower than ID or CSS selectors.

5 Link Text

- **Syntax :** link=link_text
- **Example :** link=Forgot Password
- Targets an anchor (<a>) element by its exact visible text. This is particularly useful for links with distinct text.

6 Partial link text

- **Syntax :** linkText=partial_text
- **Example :** partialLinkText=Forgot
- Similar to link text, but only requires a part of the link text. It's useful when links are long or dynamically generated.

7 Class Name

- **Syntax :** class=class_name
- **Example :** class=btn-primary
- Locates elements by their class attribute. This can be useful if the class name is unique, but often multiple elements share the same class.

Example usage in selenium IDE

Suppose you want to locate a "Login" button with the ID loginButton and verify it's present. (Fig 1)

Fig 1

```
Command: verifyElementPresent
Target: id=loginButton
```

© NIMI NOT TO BE REPUBLISHED

Location by identifier, location by id

Objective: At the end of this exercise you shall be able to
- **identify elements using their unique identifier or id attribute..**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

In Selenium IDE, locating elements by Identifier and ID are commonly used methods, though they work slightly differently. Let's break down each of these.

1 Location by ID

- **Syntax:** id=element_id
- **Example:** id=loginButton
- **Description:** This is a direct locator that targets an element with a specific id attribute. IDs are generally unique for each element on a page, making them one of the fastest and most reliable locators.
- **Use Case:** Locate elements quickly and with precision when you know the id is unique.

Example in Selenium IDE (Fig 1)

Fig 1

```
Command: click
Target: id=submitBtn
```

2 Location by Identifier

- **Syntax :** identifier=element_identifier
- **Example :** identifier=submitBtn
- **Description :** identifier is a locator type that attempts to locate an element first by its id and, if not found, then by its name attribute. This was more relevant in older versions of Selenium, where identifier offered some flexibility if the id attribute was missing.
- **Use Case :** Useful when the id might be missing, but name is present.

Example in Selenium IDE (Fig 2)

Fig 2

```
Command: type
Target: identifier=usernameField
Value: testUser
```

Key Points

- ID locators are faster and preferred if the id is unique to that element.
- Identifier locators offer some flexibility by searching for id first, then name, which can help in scenarios where an element may not have an id but has a name.

© NIMI NOT TO BE REPUBLISHED

Location by name, location by X-path

Objective: At the end of this exercise you shall be able to
- **locate elements by their "name" attribute or using xpath expressions.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

Locating elements by name and xpath in selenium ide – step-by-step guide with output

In Selenium IDE, elements can be located using different selectors. Two commonly used locators are:

By Name → Identifies elements using the name attribute.

By XPath → Locates elements using XPath expressions (useful when no other unique attribute is available).

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Start a new project or open an existing test case.

Expected Output:

"Selenium IDE launched successfully!"

"Test case ready for execution!"

Step 2 : Locating Elements by Name

Scenario : Enter text into a search box located by name.

1 Open Selenium IDE.

2 Click on Table View.

3 Add the following commands (Fig 1)

Fig 1

Command	Target	Value
open	https://example.com	
type	name=search	Selenium IDE
click	name=submit	

© NIMI NOT TO BE REPUBLISHED

Explanation

- Open → Navigates to the website.
- type → Finds an element with name="search" and enters "Selenium IDE".
- Click → Clicks the search button identified by name="submit".

Expected Log Output (Fig 2)

Fig 2

```
[PASS] Opened https://example.com
[PASS] Typed 'Selenium IDE' into element name=search
[PASS] Clicked element name=submit
[Test case executed successfully!]
```

If element is not found (Fig 3)

Fig 3

```
[FAIL] Element with name=search not found
```

Step 3: Locating Elements by XPath

Scenario: Click a login button using XPath when no name or id is available.

XPath Example:

- **Absolute XPath:** /html/body/div[1]/button (Not recommended)
- **Relative XPath:** //button[contains(text(),'Login')] (Recommended)

1 Open Selenium IDE.

2 Click on Table View.

3 Add the following commands (Fig 4)

Fig 4

Command	Target	Value
open	https://example.com	
click	xpath=//button[contains(text(),'Login')]	

Explanation

- Open → Navigates to the website.
- Click → Clicks a button with the text "Login" using XPath.

Expected Log Output (Fig 5)

Fig 5

```
[PASS] Opened https://example.com
[PASS] Clicked element xpath=//button[contains(text(),'Login')]
[Test case executed successfully!]
```

© NIMI NOT TO BE REPUBLISHED

If XPath is incorrect (Fig 6)

Fig 6

```
[FAIL] Element not found using xpath=//button[contains(text(),'Login')]
```

Location hyperlinks by link text

Objective: At the end of this exercise you shall be able to
- **identify hyperlinks using visible link text.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

In Selenium IDE, locating hyperlinks by link text is a method specifically used for finding <a> (anchor) elements based on the text displayed to users. This locator is useful for identifying links when you know the exact visible text.

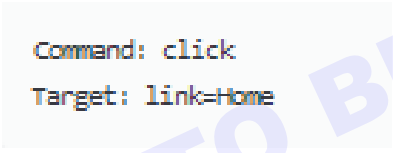
How to use link text locator

- **Command:** Use click, verifyElementPresent, or other Selenium commands with the link text locator.
- **Syntax:** link=link_text
- **Example:** link=Home

Example in Selenium IDE

If you want to click on a link with the exact visible text "Home": (Fig 1)

Fig 1



Partial Link Text

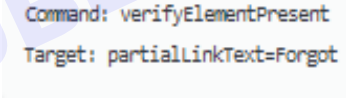
You can also locate a link by part of its visible text using partial link text. This is helpful if the link text is lengthy or partially dynamic.

- **Syntax :** partialLinkText=partial_text
- **Example :** partialLinkText=Forgot

Example in Selenium IDE (Partial Link Text)

To verify a link that contains "Forgot" in the text (like "Forgot Password?") (Fig 2)

Fig 2



Key Points

- **Exact Match: l**ink=link_text requires an exact match of the visible text in the link.
- **Case Sensitivity:** Link text is case-sensitive, so it must match the case used in the link's HTML.

© NIMI NOT TO BE REPUBLISHED

Location by DOM, Location by CSS

Objective: At the end of this exercise you shall be able to
- **identify elements using DOM structure or CSS selectors.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

Locating Elements by DOM & CSS in Selenium IDE – Step-by-Step Guide with Output

In Selenium IDE, elements on a webpage can be located using:

- DOM (Document Object Model) Location → Uses JavaScript syntax to traverse the DOM hierarchy.
- CSS Selector Location → Uses CSS patterns to find elements efficiently.

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Start a new project or open an existing test case.

Expected Output

- "Selenium IDE launched successfully!"
- "Test case ready for execution!"

Step 2: Locating Elements by DOM

DOM locators use JavaScript-like syntax:

- **Syntax :** document.forms[0].elements[2]
- **Example:** document.getElementById('login-button')

Scenario: Click a login button using DOM.

1 Open Selenium IDE.

2 Click on Table View.

3 Add the following commands (Fig 1)

Fig 1

Command	Target	Value
open	https://example.com	
click	dom=document.getElementById('login-button')	

Explanation

- open → Navigates to the website.
- click → Uses the DOM to find and click the login button.

Expected Log Output (Fig 2)

Fig 2

```
[PASS] Opened https://example.com
[PASS] Clicked element dom=document.getElementById('login-button')
[Test case executed successfully!]
```

If the element is missing (Fig 3)

Fig 3

```
[FAIL] Element not found using DOM locator
```

Step 3: Locating Elements by CSS Selectors

CSS Selectors provide a fast and flexible way to locate elements.

- CSS Selector Syntax Examples (Fig 4)

© NIMI NOT TO BE REPUBLISHED

Fig 4

Selector Type	Example	Matches
ID Selector	`css=#login-button`	Element with `id="login-button"`
Class Selector	`css=.btn-primary`	Elements with `class="btn-primary"`
Tag & Class	`css=button.btn-primary`	`<button class="btn-primary">`
Child Selector	`css=div > button`	`<div><button></button></div>`
Attribute Selector	`css=input[name='username']`	`<input name="username">`

Scenario: Enter text into a field using a CSS selector.

1 Open selenium IDE.

2 Click on table view.

3 Add the following commands (Fig 5)

Fig 5

Command	Target	Value
`open`	`https://example.com`	
`type`	`css=input[name='username']`	`testuser`

Explanation:

- open → Navigates to the website.
- type → Finds the input field with name="username" and enters "testuser".

Expected Log Output (Fig 6)

Fig 6

```
[PASS] Opened https://example.com
[PASS] Typed 'testuser' into element css=input[name='username']
[Test case executed successfully!]
```

If the CSS selector is incorrect (Fig 7)

Fig 7

```
[FAIL] Element not found using css=input[name='username']
```

© NIMI NOT TO BE REPUBLISHED

Implicit locators

Objective: At the end of this exercise you shall be able to
- **automatically locate elements using default search strategies.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

Implicit locators in selenium IDE – step-by-step guide with output

Implicit locators in selenium ide refer to the default strategy selenium uses to identify elements without explicitly specifying a locator type.

By default, selenium ide searches for elements in this order:

1 ID (id=elementID)

2 Name (name=elementName)

3 Link Text (link=LinkText)

4 XPath (xpath=//tagname[@attribute='value'])

5 CSS Selectors (css=selector)

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Start a new project or open an existing test case.

Expected Output

"Selenium IDE launched successfully!"

"Test case ready for execution!"

Step 2: Using Implicit Locators (Without Specifying Type)

Scenario: Click a "Login" button without explicitly using id=, name=, or xpath=.

1 Open Selenium IDE.

2 Click on Table View.

3 Add the following commands (Fig 1)

Fig 1

Command	Target	Value
open	https://example.com	
click	login-button	

© NIMI NOT TO BE REPUBLISHED

Explanation:

- Open → Navigates to the website.
- click → Implicitly searches for login-button using ID, Name, Link Text, XPath, and CSS (in that order).

Expected Log Output (If ID or Name Exists (Fig 2)

Fig 2

```
[PASS] Opened https://example.com
[PASS] Clicked element login-button
[Test case executed successfully!]
```

If Selenium cannot find the element (Fig 3)

Fig 3

```
[FAIL] Element 'login-button' not found using implicit locators
```

Step 3: Using implicit locators for input fields

Scenario: Enter text into a username field without specifying the locator type.

1 Open selenium IDE.

2 Click on table view.

3 Add the following commands (Fig 4)

Fig 4

Command	Target	Value
open	https://example.com	
type	username	testuser

Explanation

- type searches for username using ID, Name, Link Text, XPath, and CSS.
- If the page contains <input id="username"> or <input name="username">, it will automatically find it.

Expected Log Output (Fig 5)

Fig 5

```
[PASS] Opened https://example.com
[PASS] Typed 'testuser' into element username
[Test case executed successfully!]
```

If Selenium cannot find the element (Fig 6)

Fig 6

```
[FAIL] Element 'username' not found using implicit locators
```

© NIMI NOT TO BE REPUBLISHED

Matching text patterns

Objective: At the end of this exercise you shall be able to
- **find elements by matching specific text patterns.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

Matching Text Patterns in Selenium IDE – Step-by-Step Guide with Output

Matching Text Patterns in Selenium IDE allows you to verify or assert text on a webpage using different pattern-matching techniques, including:

- Exact Match → Checks for an exact string match.
- Glob Pattern → Uses wildcard (*) to match parts of a string.
- Regular Expressions (Regex) → Matches complex text patterns dynamically.

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.

2 Click on the Selenium IDE extension.

3 Start a new project or open an existing test case.

Expected Output:

"Selenium IDE launched successfully!"

"Test case ready for execution!"

Step 2: Matching Text with an Exact Match

Scenario: Verify a confirmation message after login.

1 Open Selenium IDE.

2 Click on Table View.

3 Add the following commands (Fig 1)

Fig 1

Command	Target	Value
open	https://example.com	
assertText	id=welcome-message	Welcome, John Doe!

Explanation

- assert text checks if the text inside the element with id="welcome-message" is exactly "Welcome, John Doe!".

Expected log Output (Fig 2)

Fig 2

```
[PASS] Opened https://example.com
[PASS] Verified text 'Welcome, John Doe!'
[Test case executed successfully!]
```

If the text does not match (Fig 3)

Fig 3

```
[FAIL] Text mismatch. Expected: 'Welcome, John Doe!' but found: 'Welcome, John'
```

Step 3: Matching Text Using a Glob Pattern (* Wildcards)

Scenario: Verify a dynamic greeting message where the name may change.

1 Open Selenium IDE.

2 Click on Table View.

3 Add the following commands (Fig 4)

© NIMI NOT TO BE REPUBLISHED

Fig 4

Command	Target	Value
open	https://example.com	
verifyText	id=welcome-message	glob:Welcome, *!

Explanation

- Verify text checks if the element id="welcome-message" starts with "Welcome, " and ends with "!" while allowing any text (*) in between.

Expected Log Output (Fig 5)

Fig 5

```
[PASS] Opened https://example.com
[PASS] Verified text 'Welcome, John!' matches 'glob:Welcome, *!'
[Test case executed successfully!]
```

If the pattern does not match (Fig 6)

Fig 6

```
[FAIL] Text mismatch. Expected pattern: 'glob:Welcome, *!' but found: 'Hello, John!'
```

Step 4: Matching Text Using Regular Expressions (Regex)

Scenario: Verify a dynamic order confirmation number.

1 Open Selenium IDE.
2 Click on Table View.
3 Add the following commands (Fig 7)

Fig 7

Command	Target	Value
open	https://example.com	
verifyText	id=order-confirmation	regexp:Order #[0-9]+ confirmed!

Explanation

- verify Text checks if id=order-confirmation contains "Order #12345 confirmed!" where 12345 can be any number ([0-9]+).

Expected Log Output (Fig 8)

Fig 8

```
[PASS] Opened https://example.com
[PASS] Verified text 'Order #56789 confirmed!' matches 'regexp:Order #[0-9]+ confirmed!'
[Test case executed successfully!]
```

If the text does not match

[FAIL] Text mismatch. Expected pattern: 'regexp:Order #[0-9]+ confirmed!' but found: 'Your order is confirmed!'

© NIMI NOT TO BE REPUBLISHED

Globbing patterns, regular expression patterns, exact patterns

Objective: At the end of this exercise you shall be able to
- **match text using wildcards, regex, or exact pattern.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

In the context of Selenium IDE, text matching can be achieved using three main types of patterns: exact patterns, globbing patterns, and regular expressions. Here's a breakdown of each:

1 Exact Patterns

- **Description:** An exact pattern matches text exactly as it appears. This is the simplest form of text matching.
- **Usage:** Use exact patterns when you want to verify that a specific piece of text is present in an element or on the page.
- **Example (Fig 1)**

Fig 1

```
Command: assertText
Target: css=.message
Value: Success
```

- This checks if the element with the class .message contains exactly the text "Success".

2 Globbing Patterns

- **Description:** Globbing patterns use wildcard characters to match a range of text values. In Selenium IDE, the * character can be used as a wildcard to represent any number of characters, while ? can represent a single character.
- **Usage:** Use globbing when you need to match text that may vary in length or characters.

Example (Fig 2)

Fig 2

```
Command: assertText
Target: css=.message
Value: Suc*
```

- This will match any text starting with "Suc", such as "Success", "Succeed", etc.

3 Regular Expression Patterns

- **Description :** Regular expressions (regex) provide a powerful way to specify complex search patterns. They allow for advanced matching criteria, such as specific sequences of characters, character classes, and quantifiers.
- **Usage :** Use regex when you need to match patterns that cannot be easily captured with exact or globbing patterns.

Example (Fig 3)

Fig 3

```
Command: assertText
Target: css=.message
Value: /Suc.*?/
```

- This regex matches any text that starts with "Suc" followed by any characters (0 or more), such as "Success" or "Succeed".

Summary Table (Fig 4)

Fig 4

Pattern Type	Description	Example
Exact Pattern	Matches text exactly	`assertText: Success`
Globbing Pattern	Uses wildcards for flexible matching	`assertText: Suc*`
Regular Expression	Advanced pattern matching with regex	`assertText: /Suc.*?/`

© NIMI NOT TO BE REPUBLISHED

The “And Wait” Commands

Objective: At the end of this exercise you shall be able to
- **perform action and wait for completion.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

In Selenium IDE, the “And Wait” commands are used to ensure that your script not only performs an action but also waits for a specific condition or state before proceeding. This is particularly useful in web automation, where elements may not be immediately available due to page load times or asynchronous content.

Common "And Wait" Commands

Here are some of the most commonly used commands with "And Wait" in Selenium IDE

1 clickAndWait

- **Description:** Clicks on an element and waits for the page to load completely before moving on to the next command.
- **Usage (Fig 1)**

Fig 1

```
Command: clickAndWait
Target: css=button#submit
```

Use case : When you want to click a button and ensure that the resulting page is fully loaded before performing any further actions.

2 selectAndWait

- **Description:** Selects an option from a dropdown and waits for the page to update or reload.
- **Usage (Fig 2)**

Fig 2

```
Command: selectAndWait
Target: css=select#dropdown
Value: optionValue
```

Use Case: Useful when selecting a dropdown option that triggers a page refresh or content update.

3 typeAndWait

- **Description:** Types text into an input field and waits for the page to reflect changes.
- **Usage (Fig 3)**

Fig 3

```
Command: typeAndWait
Target: css=input#search
Value: Selenium
```

Use Case: When you need to enter data in a form field and want to ensure that any dynamic changes triggered by the input are completed before moving on.

4 chooseFileAndWait

- **Description:** Uploads a file and waits for the action to complete.
- **Usage (Fig 4)**

Fig 4

```
Command: chooseFileAndWait
Target: css=input[type="file"]
Value: path/to/file.txt
```

- **Use Case:** When uploading files, and you need to ensure that the upload is completed before the next command runs.

© NIMI NOT TO BE REPUBLISHED

The wait for commands in AJAX applications

Objective: At the end of this exercise you shall be able to
- **wait for AJAX elements to load or change.**

Requirements

Tools/Equipments/Machines
- A working PC with IDE software - as reqd.

PROCEDURE

In AJAX applications, where content is dynamically loaded and may not be immediately available after an action (like a button click), using wait commands is crucial to ensure that your Selenium IDE tests run smoothly and reliably. Selenium provides various commands to wait for specific conditions before proceeding with the next step. Here's an overview of the main wait for commands you can use in Selenium IDE:

1 waitForElementPresent
- **Description:** Waits until a specified element is present in the DOM.
- **Usage (Fig 1)**

Fig 1
```
Command: waitForElementPresent
Target: css=.dynamic-content
```

- **Use Case:** Use this command when you expect an element to appear after an AJAX call, ensuring that the script only continues when the element is present.

2 waitForVisible
- **Description:** Waits until a specified element is visible on the page.
- **Usage (Fig 2)**

Fig 2
```
Command: waitForVisible
Target: css=.loading-spinner
```

- **Use Case:** Ideal for scenarios where you want to ensure that a loading spinner or any other element is visible before proceeding.

3 waitForText
- **Description:** Waits until a specified element contains a specific text.
- **Usage (Fig 3)**

Fig 3
```
Command: waitForText
Target: css=.message
Value: "Data loaded successfully"
```

- **Use Case:** Use this command when you expect a specific message or notification to appear after an AJAX request completes.

4 waitForCondition
- **Description:** Waits for a JavaScript condition to be true.
- **Usage (Fig 4)**

Fig 4
```
Command: waitForCondition
Target: "selenium.browserbot.getCurrentWindow().document.readyState == 'complete'"
```

- **Use case:** Useful for waiting for complex conditions, like waiting for all AJAX requests to finish or ensuring that the page has fully loaded.

5 waitForElementNotPresent
- **Description:** Waits until a specified element is no longer present in the DOM.
- **Usage (Fig 5)**

Fig 5
```
Command: waitForElementNotPresent
Target: css=.loading-spinner
```

- **Use Case:** Use this to wait for a loading spinner or overlay to disappear, indicating that the content has finished loading.

© NIMI NOT TO BE REPUBLISHED

Summary of usage

Command	Description	Use Case
waitForElement Present	Waits for an element to appear in the DOM	When elements are dynamically added after AJAX
waitForVisible	Waits for an element to become Visible	Ensuring that loading indicators are finished
waitForText	Waits for a specific text to appear in an element	Verifying success messages after actions
waitForCondition	Waits for a custom JavaScript condition	More complex wait conditions
waitForElementNotPresent	Waits for an element to disappear	Ensuring that loading indicators are gone

Sequence of evaluation and flow control

Objective: At the end of this exercise you shall be able to
- **control test flow and evaluate conditions in sequence.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

Sequence of evaluation and flow control in selenium ide – step-by-step guide with output

Flow control in Selenium IDE allows test cases to execute conditionally, iterate through loops, and handle execution sequences dynamically.

Key flow control commands:

- if/elseIf/else → Conditional Execution
- while → Loops until a condition is met
- forEach → Iterates over a list of elements
- times → Runs a block of code multiple times
- do...while → Executes at least once, then checks the condition
- store/executeScript → Stores and manipulates variables

Step 1: Open Selenium IDE

1 Open Google Chrome or Mozilla Firefox.
2 Click on the Selenium IDE extension.
3 Start a new project or open an existing test case.

Expected output

"Selenium IDE launched successfully!"

"Test case ready for execution!"

Step 2: Using if Statements for Conditional Execution

Scenario: Click "Login" only if the "username" field is visible.

1 Open selenium IDE.
2 Click on Table View.
3 Add the following commands (Fig 1)

Fig 1

Command	Target	Value
open	https://example.com/login	
if	selenium.isElementPresent("id=username")	
type	id=username	testuser
click	id=login-button	
end		

Explanation

- If checks if id=username exists before entering text.
- Type enters "testuser" only if the field exists.
- Click clicks "login-button" if "username" is present.

Expected Log Output (Fig 2)

Fig 2

```
[PASS] Opened https://example.com/login
[PASS] Username field is present
[PASS] Entered 'testuser'
[PASS] Clicked login button
[Test case executed successfully!]
```

© NIMI NOT TO BE REPUBLISHED

If the username field is missing (Fig 3)

Fig 3

```
[PASS] Opened https://example.com/login
[FAIL] Username field is not present. Skipping login.
```

Step 3: Using while Loops to Wait for a Condition

Scenario: Wait until a loading spinner disappears before proceeding. (Fig 4)

Fig 4

Command	Target	Value
open	https://example.com/dashboard	
while	selenium.isElementPresent("id=loading-spinner")	
pause	1000	
end		
click	id=start-button	

Explanation

- While keeps checking for "loading-spinner".
- Pause waits 1 second before checking again.
- Click proceeds once the spinner disappears.

Expected Log Output (Fig 5)

Fig 5

```
[PASS] Opened https://example.com/dashboard
[PASS] Loading spinner detected, waiting...
[PASS] Loading spinner disappeared. Proceeding.
[PASS] Clicked start button
[Test case executed successfully!]
```

© NIMI NOT TO BE REPUBLISHED

If the spinner never disappears (Fig 6)

Fig 6

```
[FAIL] Timed out waiting for loading-spinner to disappear.
```

Step 4: Using for Each to Iterate Over Multiple Elements

Scenario: Click on each item in a list. (Fig 7)

Fig 7

Command	Target	Value
open	https://example.com/products	
forEach	css=.product-item	product
click	storedVars['product']	
end		

Explanation

- for Each finds all elements matching css=.product-item.
- click clicks each product-item one by one.

Expected Log Output (Fig 8)

Fig 8

```
[PASS] Opened https://example.com/products
[PASS] Found 5 product items
[PASS] Clicked product #1
[PASS] Clicked product #2
[PASS] Clicked product #3
[PASS] Clicked product #4
[PASS] Clicked product #5
[Test case executed successfully!]
```

© NIMI NOT TO BE REPUBLISHED

Step 5: Using times to Repeat Steps

Scenario: Refresh the page 3 times before checking for an element. (Fig 9)

Fig 9

Command	Target	Value
open	https://example.com/loading	
times	3	
refresh		
pause	2000	
end		
assertElementPresent	id=finished-loading	

Explanation

- times runs the refresh-pause sequence 3 times before checking if "finished-loading" is present.

Expected Log Output (Fig 10)

Fig 10

```
[PASS] Opened https://example.com/loading
[PASS] Refreshed page (Attempt 1)
[PASS] Refreshed page (Attempt 2)
[PASS] Refreshed page (Attempt 3)
[PASS] Verified 'finished-loading' is present
[Test case executed successfully!]
```

If the element never appears (Fig 11)

Fig 11

```
[FAIL] Element 'finished-loading' not found after 3 refresh attempts
```

Step 6: Using do...while for Post-Condition Loops

Scenario: Keep scrolling until the "Load More" button disappears. (Fig 12)

Fig 12

Command	Target	Value
open	https://example.com/articles	
do		
click	id=load-more	
pause	2000	
while	selenium.isElementPresent("id=load-more")	

Explanation

- do ensures the script clicks "Load More" at least once.
- while keeps clicking "Load More" until it disappears.

Expected Log Output (Fig 13)

Fig 13

```
[PASS] Opened https://example.com/articles
[PASS] Clicked 'Load More' (Attempt 1)
[PASS] Clicked 'Load More' (Attempt 2)
[PASS] No more 'Load More' button found. Stopping.
[Test case executed successfully!]
```

If an error occurs while clicking (Fig 14)

Fig 14

```
[FAIL] Unable to click 'Load More'. Element not interactable.
```

© NIMI NOT TO BE REPUBLISHED

Store commands and selenium variables

Objective: At the end of this exercise you shall be able to
- **store values and reuse them using selenium variables.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

In Selenium IDE, store commands and Selenium variables are used to save values during test execution, which can then be reused later in the test. This is particularly useful for handling dynamic data or passing values between different commands. Here's a comprehensive overview:

Store Commands

Store commands allow you to save data or results into variables for later use. Some of the commonly used store commands include:

1 store

- **Description:** Stores a specific value in a variable.
- **Usage (Fig 1)**

Fig 1

```
Command: store
Target: Some value
Value: myVariable
```

- **Example (Fig 2)**

Fig 2

```
Command: store
Target: "Selenium"
Value: testName
```

2 storeText

- **Description:** Stores the visible text of a specified element.
- **Usage (Fig 3)**

Fig 3

```
Command: storeText
Target: css=.elementSelector
Value: myTextVariable
```

- **Example (Fig 4)**

Fig 4

```
Command: storeText
Target: css=h1
Value: headerText
```

3 storeValue

- **Description:** Stores the value of an input field or any element.
- **Usage (Fig 5)**

Fig 5

```
Command: storeValue
Target: css=input#username
Value: usernameValue
```

- **Example (Fig 6)**

Fig 6

```
Command: storeValue
Target: css=input#email
Value: emailValue
```

4 storeElementCount

- **Description:** Stores the number of elements matching a specified locator.
- **Usage (Fig 7)**

Fig 7

```
Command: storeElementCount
Target: css=.items
Value: itemCount
```

© NIMI NOT TO BE REPUBLISHED

- **Example (Fig 8)**

Fig 8

```
Command: storeElementCount
Target: css=.product
Value: numberOfProducts
```

Using selenium variables

Selenium variables allow you to store and retrieve values dynamically during the execution of your tests. You can use these variables to refer to stored data throughout your script.

Example of using variables

Here's how you might use store commands and variables in a Selenium IDE script. (Fig 9)

Accessing stored variables

To access the stored values in your script, use the ${variableName} syntax. For example, if you stored a welcome message in a variable called welcomeText, you can reference it as ${welcomeText} in subsequent commands, as shown in the echo command above.

Variable Scope

- **Local Variables:** Variables created within a specific test case are local to that case and cannot be accessed outside of it.
- **Global Variables:** You can create global variables by storing them at a higher scope, allowing access across multiple test cases in a suite.

Fig 9

```
Command: open
Target: http://example.com/login

Command: type
Target: css=input#username
Value: myUsername

Command: type
Target: css=input#password
Value: myPassword

Command: click
Target: css=button#login

Command: waitForElementPresent
Target: css=.welcomeMessage

Command: storeText
Target: css=.welcomeMessage
Value: welcomeText

Command: echo
Target: "Welcome message: ${welcomeText}"
```

© NIMI NOT TO BE REPUBLISHED

Store element present, store text, store evaluation

Objective: At the end of this exercise you shall be able to
- **store element presence, text, or evaluation results for later use.**

Requirements

Tools/Equipments/Machines

- A working PC with IDE software - as reqd.

PROCEDURE

Selenium IDE is a record-and-playback tool for web application testing. It allows testers to automate browser interactions using a simple interface. In this guide, I'll explain how to use the following commands in Selenium IDE with step-by-step instructions and expected outputs

- Store Element Present
- Store Text
- Store Eval

1 storeElementPresent

This command checks whether a specific element is present on the webpage and stores the result (true or false) in a variable.

Steps

1 Open Selenium IDE and start recording or create a new test.

2 Add the command manually

- **Command:** store Element Present
- **Target:** Locator of the element (e.g., id=loginButton)
- **Value:** elementExists

3 Run the test.

Example (Fig 1)

Fig 1

Command	Target	Value
storeElementPresent	id=loginButton	elementExists

Expected Output

- If the element is present, elementExists will store true.
- If the element is absent, elementExists will store false.

Verification: To verify, add a echo command (Fig 2)

Fig 2

Command	Target	Value
echo	${elementExists}	

Output (Displayed in the Log Panel) (Fig 3 & 4)

Fig 3

```
true
```

Or

Fig 4

```
false
```

2 storeText

This command extracts the text from an element and stores it in a variable.

Steps

1 Identify the element whose text you want to extract.

2 Add the following command:

- **Command:** storeText
- **Target:** Locator of the element (e.g., id=welcomeMessage)
- **Value:** storedText

© NIMI NOT TO BE REPUBLISHED

3 Run the test

Example (Fig 5)

Fig 5

Command	Target	Value
storeText	id=welcomeMessage	storedText

Expected Output (Fig 6)

Fig 6

Hello, User!

If the element Hello, User! exists, storedText will contain.

Verification (Fig 7)

Fig 7

Command	Target	Value
echo	${storedText}	

Output (Fig 8)

Fig 8

Hello, User!

3 storeEval

This command allows executing JavaScript expressions and storing the result in a variable.

Steps

1 Add the command

- **Command:** storeEval
- **Target:** JavaScript code (e.g., 2 + 2)
- **Value:** result

2 Run the test

Example (Fig 9)

Fig 9

Command	Target	Value
storeEval	2 + 2	result

Expected Output

result will store (Fig 10)

Fig 10

4

Verification (Fig 11)

Fig 11

Command	Target	Value
echo	${result}	

Output (Fig 12)

Fig 12

4

© NIMI NOT TO BE REPUBLISHED

JavaScript and selenese parameters

Objective : At the end of this exercise you shall be able to
- **pass parameters in javascript and selenese commands.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

In Selenium IDE, **Selenese** is the language used to write test scripts, and it consists of commands that interact with web elements. When working with Selenese, you can also integrate **JavaScript** for more complex operations or to handle dynamic situations. Both Selenese and JavaScript can accept **parameters** that are essential for their functionality. Here's a detailed overview of how these parameters work in both contexts:

Selenese Parameters

Selenese commands typically accept three parameters:

1. **Command:** The action you want to perform (e.g., click, type, storeText).
2. **Target:** The locator for the element you want to interact with (e.g., CSS selector, XPath).
3. **Value:** The additional data or input needed for the command (used mainly with commands like type, store, etc.).

Example of Selenese Parameters

Here's a simple example using Selenese parameters. (Fig 1)

Fig 1

```
Command: open
Target: http://example.com/login

Command: type
Target: css=input#username
Value: myUsername

Command: type
Target: css=input#password
Value: myPassword

Command: click
Target: css=button#login
```

In this example:

- **Command:** open, type, click
- **Target:** URLs, CSS selectors for elements
- **Value:** Strings like myUsername and myPassword

JavaScript Parameters in Selenium IDE

When using JavaScript within Selenium IDE, especially with commands like storeEval, you can pass parameters to JavaScript functions. This allows for more complex operations, data manipulation, or calculations.

Example of JavaScript with Parameters (Fig 2)

Fig 2

```
Command: storeEval
Target: "function add(a, b) { return a + b; } add(5, 10);"
Value: sum
```

In this example, we define a JavaScript function add, which takes two parameters (a and b). The command will execute this function with the parameters 5 and 10, and store the result in the variable sum.

Common Use Cases for Parameters

1. **Dynamic Input:** You can use variables in place of hardcoded values in both Selenese and JavaScript. For example. (Fig 3)

Fig 3

```
Command: type
Target: css=input#username
Value: ${usernameVariable}
```

2. **Function Parameters in JavaScript:** You can create reusable JavaScript functions that accept parameters to perform different tasks based on input. (Fig 4)

Fig 4

```
Command: storeEval
Target: "function multiply(x, y) { return x * y; } multiply(${num1}, ${num2});"
Value: product
```

© NIMI NOT TO BE REPUBLISHED

3 **Conditional Logic:** You can use JavaScript to perform checks and balances in your tests. For instance. (Fig 5)

Fig 5

```
Command: storeEval
Target: "return document.title === 'Expected Title';"
Value: isTitleCorrect
```

Summary of Selenese and JavaScript Parameters. (Fig 6)

Fig 6

Aspect	Selenese	JavaScript
Type of Parameters	Command, Target, Value	Function parameters (e.g., `a`, `b`)
Usage	Directly interacts with web elements	Performs calculations or manipulations
Example	`Command: type	Target: css=#input

© NIMI
NOT TO BE REPUBLISHED

JavaScript usages with script parameters

Objective : At the end of this exercise you shall be able to
- **use javascript to handle dynamic script parameters.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Using JavaScript in conjunction with script parameters in Selenium IDE allows for more dynamic and flexible test scripts. This integration enables you to perform calculations, manipulate data, and execute complex logic that might not be easily achievable with Selenese alone. Below are various usages of JavaScript with script parameters in Selenium IDE.

Common Usages of JavaScript with Script Parameters

1 Defining Functions with Parameters

You can define JavaScript functions that accept parameters, which allows you to reuse code and perform operations based on different inputs. (Fig 1)

Fig 1

```
Command: storeEval
Target: "function add(a, b) { return a + b; } add(5, 10);"
Value: sum
```

In this example, a function add is defined, taking two parameters a and b, and returning their sum. The result is stored in the variable sum.

2 Using Variables in JavaScript Functions

You can pass Selenium variables to JavaScript functions for dynamic execution. This is useful for incorporating test data directly into your calculations. (Fig 2)

Fig 2

```
Command: store
Target: "5"
Value: num1

Command: store
Target: "10"
Value: num2

Command: storeEval
Target: "function multiply(x, y) { return x * y; } multiply(${num1}, ${num2});"
Value: product
```

Here, the JavaScript function multiply takes two parameters (from Selenium variables num1 and num2) and calculates their product.

3 Conditional Logic in JavaScript

JavaScript can also handle conditional statements, allowing you to make decisions based on parameters. (Fig 3)

Fig 3

```
Command: store
Target: "20"
Value: threshold

Command: storeEval
Target: "var value = 25; value > ${threshold} ? 'Above threshold' : 'Below threshold';"
Value: comparisonResult
```

This example checks if a hardcoded value (25) is above a dynamically stored threshold and returns a corresponding message.

4 Retrieving DOM Elements with Parameters

You can use JavaScript to manipulate or retrieve values from the DOM, using parameters to specify which elements to interact with.In this case, the JavaScript retrieves the text from an element with a specific ID and stores it in the variable elementText. (Fig 4)

Fig 4

```
Command: storeEval
Target: "return document.getElementById('dynamicElementId').innerText;"
Value: elementText
```

© NIMI NOT TO BE REPUBLISHED

Example Scenario: Full Test Script

Here's a more comprehensive example incorporating various usages of JavaScript with parameters in a Selenium IDE test script. (Fig 5)

Fig 5

```
Command: open
Target: http://example.com/form

Command: type
Target: css=input#username
Value: myUsername

Command: type
Target: css=input#password
Value: myPassword

Command: click
Target: css=button#submit

Command: waitForElementPresent
Target: css=.resultMessage

Command: storeText
Target: css=.resultMessage
Value: resultMessage

Command: storeEval
Target: "function analyzeResult(message) { return message.includes('success') ? 'Test Passed' :
Value: testResult
```

© NIMI NOT TO BE REPUBLISHED

JavaScript usage with non-script parameters

Objective : At the end of this exercise you shall be able to
- **integrate javascript to manage non-script parameters dynamically.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

In Selenium IDE, non-script parameters refer to values that are hardcoded or predefined, rather than dynamically generated within the script. When using JavaScript in conjunction with non-script parameters, you can leverage static values directly within your JavaScript code. Here's a detailed overview of how to use JavaScript with non-script parameters in Selenium IDE.

Common Usages of JavaScript with Non-Script Parameters

1 Basic Arithmetic Operations

You can perform arithmetic calculations directly using non-script parameters within JavaScript. (Fig 1)

Fig 1

```
Command: storeEval
Target: "var result = 10 + 20; result;"
Value: sum
```

Here, the sum of two hardcoded numbers (10 and 20) is calculated and stored in the variable sum.

2 String Manipulation

You can manipulate string literals using JavaScript functions. (Fig 2)

In this example, two non-script string literals are concatenated, and the result is stored in the variable full Greeting.

Fig 2

```
Command: storeEval
Target: "var greeting = 'Hello'; var name = 'World'; greeting + ' ' + name;"
Value: fullGreeting
```

3 Conditional Logic with Hardcoded Values

You can use non-script parameters in conditional statements to execute logic based on static values. (Fig 3)

Fig 3

```
Command: storeEval
Target: "var score = 85; score >= 50 ? 'Pass' : 'Fail';"
Value: result
```

This script checks a hardcoded score and returns either "Pass" or "Fail" based on the value, storing the result in the variable result.

4 Working with Arrays

You can define arrays with static values and perform operations on them. (Fig 4)

Fig 4

```
Command: storeEval
Target: "var fruits = ['apple', 'banana', 'cherry']; fruits.length;"
Value: fruitCount
```

In this example, an array of hardcoded fruit names is created, and the count of elements in the array is stored in the variable fruit Count.

5 Using Non-Script Parameters in Functions

You can define functions that take hardcoded values as parameters.

This example defines a function calculate Area that multiplies two hardcoded values (length and width) and stores the calculated area in the variable area. (Fig 5)

Fig 5

```
Command: storeEval
Target: "function calculateArea(length, width) { return length * width; } calculateArea(5, 1
Value: area
```

© NIMI NOT TO BE REPUBLISHED

Example Scenario

Here's a full example that integrates JavaScript with non-script parameters within a Selenium IDE test script. (Fig 6)

Fig 6

```
Command: open
Target: http://example.com

Command: storeEval
Target: "var base = 5; var height = 10; var area = (base * height) / 2; area;"
Value: triangleArea

Command: echo
Target: "The area of the triangle is: ${triangleArea}"

Command: storeEval
Target: "var fruitArray = ['apple', 'banana', 'orange']; fruitArray.join(', ');"
Value: fruitList

Command: echo
Target: "Fruits: ${fruitList}"

Command: storeEval
Target: "var temperature = 30; temperature > 25 ? 'Hot' : 'Cold';"
Value: weatherStatus
```

© NIMI
NOT TO BE REPUBLISHED

Echo - the selenese print commands

Objective: At the end of this exercise you shall be able to
- **print messages to the console using selenese.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Selenium IDE is a popular tool for automating web applications for testing purposes. Here's a guide on how to use the echo command and other basic commands in Selenium IDE, particularly for your requirement regarding "The Selenese Print":

1 Echo Command

The echo command is used to display a message in the log output. This is useful for debugging or for providing information during test execution.

Syntax (Fig 1)

Fig 1
```
echo | <message>
```

Example (Fig 2)

Fig 2
```
echo | "Starting the test execution"
```

2 Basic Commands in Selenium IDE

a Open Command

This command opens a specified URL in the brows. (Fig 3)

Fig 3
```
Command: open
Target: http://example.com
```

b Click Command

This command clicks on an element, specified by a locator. (Fig 4)

Fig 4
```
Command: click
Target: css=button#submit
```

c Type Command

This command types text into an input field. (Fig 5)

Fig 5
```
Command: type
Target: id=username
Value: testuser
```

d Assert Command

This command checks if the given condition is true. (Fig 6)

Fig 6
```
Command: assert
Target: css=h1
Value: "Welcome"
```

e Wait Command

This command pauses the test execution for a specified amount of time. (Fig 7)

Fig 7
```
Command: waitForElementPresent
Target: css=.loading
```

3 Creating a Sample Test Script

Here's a simple example of how to combine these commands in a test case. (Fig 8)

Fig 8
```
Command: open
Target: http://example.com

Command: echo
Target: "Navigating to the homepage"

Command: click
Target: css=button#login

Command: echo
Target: "Clicked on the login button"

Command: type
Target: id=username
Value: testuser

Command: type
Target: id=password
Value: secretpassword

Command: click
Target: css=button#submit

Command: waitForElementPresent
Target: css=.welcome-message

Command: echo
Target: "Login successful, welcome message displayed"
```

4 Running the Test

To run your test, simply play it in the Selenium IDE interface. The output from the echo commands will appear in the log, which helps track the flow of execution and debug if necessary.

© NIMI NOT TO BE REPUBLISHED

Alert, popups and multiple windows

Objective : At the end of this exercise you shall be able to
- **handle alerts, popups, and switch between windows.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Handling alerts, pop-ups, and multiple windows is an essential aspect of web automation testing using Selenium IDE. Below are instructions on how to manage these elements effectively.

1 Handling Alerts

Alerts are JavaScript pop-ups that require user interaction. Selenium IDE provides commands to manage them.

a Accepting an Alert

To accept (click "OK") an alert. (Fig 1)

Fig 1

```
Command: alert
Target: accept
```

b Dismissing an Alert (Fig 2)

Fig 2

```
Command: alert
Target: dismiss
```

c Getting Alert Text (Fig 3)

To retrieve the text from an alert.

Fig 3

```
Command: alert
Target: getText
```

d Example of Handling an Alert (Fig 4)

Fig 4

```
Command: click
Target: css=button#trigger-alert  // Click a button that triggers an alert

Command: alert
Target: accept  // Accept the alert
```

2 Handling Pop-ups

Pop-ups can include prompts and confirmation boxes. You can handle these similarly to alerts.

a Prompting for Input

To input text in a prompt. (Fig 5)

Fig 5

```
Command: alert
Target: sendKeys
Value: "Sample Input"  // Provide input to the prompt
```

3 Handling Multiple Windows

When dealing with multiple windows, you may need to switch between them. Selenium IDE has specific commands for this.

a Switching to a Window

To switch to another window, you typically need the window title or the index. (Fig 6)

Fig 6

```
Command: selectWindow
Target: name=windowName  // Switch to the specified window by name
```

b Closing a Window

To close the current window. (Fig 7)

Fig 7

```
Command: close
```

© NIMI NOT TO BE REPUBLISHED

c Switching Back to the Main Window

After performing actions in a pop-up, you may want to return to the main window. (Fig 8)

Fig 8

```
Command: selectWindow
Target: name=mainWindowName  // Return to the main window
```

d Example of Handling Multiple Windows (Fig 9)

Fig 9

```
Command: click
Target: css=button#open-new-window  // Click a button that opens a new window

Command: selectWindow
Target: title=New Window Title  // Switch to the new window

Command: echo
Target: "Now in the new window"

Command: close  // Close the new window

Command: selectWindow
Target: name=mainWindow  // Switch back to the main window
```

© NIMI NOT TO BE REPUBLISHED

Alerts, Confirmations

Objective: At the end of this exercise you shall be able to
- **handle alert and confirmation dialogs in selenium.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Handling alerts and confirmation dialogs is crucial for automating web applications using Selenium IDE. Below is a detailed guide on how to work with alerts and confirmations, including command examples.

1 Alerts

Alerts are simple pop-up messages that require user interaction. They typically display a message and provide an "OK" button.

a Accepting an Alert

To accept an alert (click "OK"), use the alert command with the accept target.

Example (Fig 1)

Fig 1

```
Command: click
Target: css=button#trigger-alert  // Click a button that triggers the alert

Command: alert
Target: accept  // Accept the alert
```

b Dismissing an Alert

If you want to dismiss the alert (click "Cancel"), you can use the dismiss action.

Example (Fig 2)

Fig 2

```
Command: click
Target: css=button#trigger-confirm  // Click a button that triggers the confirmation alert

Command: alert
Target: dismiss  // Dismiss the alert
```

c Getting Alert Text

To retrieve the text displayed in the alert, use the get Text action.

Example (Fig 3)

Fig 3

```
Command: click
Target: css=button#trigger-alert

Command: alert
Target: getText  // Get the alert text

Command: echo
Target: "${alertText}"  // Display the alert text in the log
```

2 Confirmation Dialogs

Confirmation dialogs are similar to alerts, but they usually require a choice between "OK" and "Cancel."

a Accepting a Confirmation

To accept a confirmation dialog, use the accept command.

Example (Fig 4)

Fig 4

```
Command: click
Target: css=button#confirm-action  // Click a button that triggers a confirmation

Command: alert
Target: accept  // Accept the confirmation dialog
```

© NIMI NOT TO BE REPUBLISHED

b Dismissing a Confirmation

To dismiss a confirmation dialog, use the dismiss command.

Example (Fig 5)

Fig 5

```
Command: click
Target: css=button#confirm-action  // Click a button that triggers a confirmation

Command: alert
Target: dismiss  // Dismiss the confirmation dialog
```

3 Full Example

Here's a complete example that demonstrates how to handle both alerts and confirmation dialogs in a single test case. (Fig 6)

Fig 6

```
Command: open
Target: http://example.com  // Navigate to the target website

Command: click
Target: css=button#trigger-alert  // Click a button to trigger an alert

Command: alert
Target: getText  // Get alert text
Command: echo
Target: "Alert text is: ${alertText}"  // Log alert text

Command: alert
Target: accept  // Accept the alert

Command: click
Target: css=button#trigger-confirm  // Click a button to trigger a confirmation

Command: alert
Target: dismiss  // Dismiss the confirmation dialog

Command: echo
Target: "Confirmation dialog dismissed."
```

© NIMI NOT TO BE REPUBLISHED

Debugging

Objective: At the end of this exercise you shall be able to
- **identify and fix issues in test scripts.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Debugging in Selenium IDE – Step by Step Guide with Output

Debugging is essential in **Selenium IDE** to identify and fix issues in automated test scripts. Selenium IDE provides built-in debugging features like **breakpoints, step execution, and logging** to help testers troubleshoot errors effectively.

Steps to Debug a Selenium IDE Script:

Step 1: Open Selenium IDE

- Launch the **Selenium IDE** extension in your browser.
- Open an existing test case or create a new one.

Step 2: Add Breakpoints

A **breakpoint** pauses execution at a specific command, allowing you to inspect the state of the script.

How to Add a Breakpoint:

1 Right-click on any command in the Test Script Table.

2 Select "Toggle Breakpoint".

3 A red dot appears next to the command, indicating a breakpoint.

Example (Fig 1)

Fig 1

Command	Target	Value
`open`	`https://example.com`	
`click`	`id=loginButton`	
`storeText`	`id=welcomeText`	`message`

Expected Output in Log Panel

When the script reaches the click command, execution will **pause**. **(Fig 2)**

Fig 2

```
[INFO] Paused at command: click | id=loginButton
```

Step 3: Use the Step Execution Feature

After hitting a breakpoint, step execution lets you execute the script one step at a time.

How to Step Through Commands

1 Click "Resume" (▶) to continue running the script.

2 Click "Step Over" (■) to execute the next command one step at a time.

3 Observe changes in the Variables Panel and Log Panel.

Expected Output

Each executed command will be logged. (Fig 3)

Fig 3

```
[INFO] Executing click | id=loginButton
[INFO] Successfully clicked loginButton
```

Step 4: Use Log Panel for Error Debugging

The **Log Panel** records every command execution and errors encountered.

Example

If an element is missing, the Log Panel may show. (Fig 4)

Fig 4

```
[ERROR] Element not found: id=loginButton
```

© NIMI NOT TO BE REPUBLISHED

Fix: Verify the locator (ID, XPath, CSS Selector) and correct it.

Step 5: Check Variables Panel

The Variables Panel stores values assigned during execution.

Example (Fig 5)

Fig 5

Command	Target	Value
`storeText`	`id=welcomeText`	`message`

If the welcome Text element contains "Welcome, John!", then (Fig 6)

Fig 6

```
[INFO] Stored message = Welcome, John!
```

Fix: If the value is incorrect, check the element's locator.

Step 6: Resume or Stop Execution

- Click "Resume" (▶) to continue execution until completion.
- Click "Stop" (■) if an issue needs to be fixed before re-running.

© NIMI NOT TO BE REPUBLISHED

Breakpoint and start points

Objective: At the end of this exercise you shall be able to
- **pause execution at specific points for debugging.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Breakpoints and Start Points in Selenium IDE – Step by Step Guide with Output

In **Selenium IDE, Breakpoints** and **Start Points** help control test execution for debugging purposes.

- **Breakpoints:** Pause execution at a specific command for debugging.
- **Start Points:** Begin execution from a specific command instead of running from the start.

Step-by-Step Guide

1 Open Selenium IDE

- Open the Selenium IDE extension in your browser.
- Load an existing test case or record a new one.

Setting a Breakpoint (Pausing Execution)

A **Breakpoint** allows you to **pause execution** at a specific command to inspect values and check issues.

Steps to Add a Breakpoint:

1 **Right-click** on the command where you want to pause execution.

2 Select **"Toggle Breakpoint"** from the menu.

3 A **red dot** appears next to the command, indicating a breakpoint.

Example (Fig 1)

Fig 1

Command	Target	Value
`open`	`https://example.com`	
`click`	`id=loginButton`	
`storeText`	`id=welcomeText`	`message`

Expected Output in Log Panel (Fig 2)

Fig 2

```
[INFO] Paused at command: click | id=loginButton
```

Debugging Actions

- Use **Step Over** (■) to run commands one by one.
- Check the **Log Panel** and **Variables Panel** for errors.
- Click **Resume** (▶) to continue execution.

Setting a Start Point (Starting Execution from a Specific Step)

A **Start Point** allows you to **begin execution from a specific command** instead of running the entire test from the start.

Steps to Set a Start Point

1 **Right-click** on the command from where you want to start execution.

2 Select **"Set Start Point"** from the menu.

3 A **green arrow** appears next to the command, indicating a **Start Point.**

© NIMI NOT TO BE REPUBLISHED

Example (Fig 3)

Fig 3

Command	Target	Value
`open`	`https://example.com`	
`click`	`id=loginButton`	
`storeText`	`id=welcomeText`	`message`

Expected Output in Log Panel (Fig 4)

Fig 4

```
[INFO] Execution started from command: storeText | id=welcomeText
```

Advantages of Start Points

- Avoid re-running the full test from the beginning.
- Useful for debugging later steps in long test scripts.

© NIMI
NOT TO BE REPUBLISHED

Stepping through a test case

Objective: At the end of this exercise you shall be able to
- **execute test case step-by-step for debugging.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Stepping Through a Test Case in Selenium IDE – Step-by-Step Guide with Output

Stepping through a test case in **Selenium IDE** helps execute commands **one at a time,** making it easier to debug issues. Instead of running the entire script at once, you can inspect each step, verify element interactions, and ensure the expected behavior.

Step-by-Step Guide to Stepping Through a Test Case

1 Open Selenium IDE and Load Your Test Case

- Open Selenium IDE in your browser.
- Load an existing test case or record a new one.

2 Add a Breakpoint (Pause Execution at a Specific Step)

- **Breakpoints** pause execution at a specific command, allowing you to step through the test manually.
- To add a breakpoint
  1 **Right-click** on any command.
  2 Select **"Toggle Breakpoint"**.
  3 A **red dot** appears next to the command, indicating a breakpoint. (Fig 1)

Fig 1

Command	Target	Value
`open`	`https://example.com`	
`click`	`id=loginButton`	
`storeText`	`id=welcomeText`	`message`

Example Test Script (Fig 2)

Fig 2

```
[INFO] Paused at command: click | id=loginButton
```

Expected Output in Log Panel

3 Start Execution and Step Through Commands

- Click **"Play Current Test Case"** (▶) to start execution.
- When the breakpoint is reached, the script will **pause**.
- Use **Step Execution** to move forward **one command at a time.**

How to Step Through Execution (Fig 3)

Fig 3

Button	Function
⏭ Step Over	Executes the next command and pauses.
▶ Resume	Runs the rest of the test after the breakpoint.
■ Stop	Stops execution if needed.

© NIMI NOT TO BE REPUBLISHED

4 Verify Outputs in the Log and Variables Panel

- **Log Panel** displays executed commands, errors, and messages.
- **Variables Panel** shows stored values, useful for debugging.

Example Output in Log Panel (Fig 4)

Fig 4

```
[INFO] Executing click | id=loginButton
[INFO] Successfully clicked loginButton
```

If an error occurs, it might display (Fig 5)

Fig 5

```
[ERROR] Element not found: id=loginButton
```

Fix: Check the element's locator (ID, XPath, CSS Selector) and correct it.

5 Resume or Stop Execution

- Click ▶ **Resume** to complete the test.
- Click ■ **Stop** if debugging is complete.

© NIMI
NOT TO BE REPUBLISHED

Find button

Objective: At the end of this exercise you shall be able to
- **locate and interact with a button element.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

In Selenium IDE, locating a button (or any web element) is crucial for automating interactions such as clicking, checking, or asserting its presence. To find a button, you'll typically use various locator strategies. Below, I'll explain how to find a button and provide examples using different locator types.

Common Locator Strategies

1 ID Selector

- If the button has a unique ID attribute, this is often the simplest and most efficient way to locate it.
- **Example (Fig 1)**

Fig 1

```
Command: click
Target: id=submit-button  // Replace with the actual ID of the button
```

2 CSS Selector

- CSS selectors allow you to target elements using CSS syntax, which can be more flexible than ID selectors.
- **Example (Fig 2)**

Fig 2

```
Command: click
Target: css=button.submit  // Use a class name, tag name, or more complex CSS selectors
```

3 XPath

- XPath is a powerful way to locate elements based on their position in the HTML structure. This is useful if the button does not have a unique identifier.
- **Example (Fig 3)**

Fig 3

```
Command: click
Target: name=submit  // Replace with the actual name of the button
```

4 Name Attribute

- If the button has a name attribute, you can also locate it using that.
- **Example (Fig 4)**

Fig 4

```
Command: click
Target: name=submit  // Replace with the actual name of the button
```

© NIMI NOT TO BE REPUBLISHED

5 Link Text

- If the button is an anchor (<a> tag), you can use the text inside the link.
- **Example (Fig 5)**

Fig 5

```
Command: click
Target: link=Click Here  // Use the visible text of the link/button
```

Example of Finding a Button

Let's say you want to find and click a button on a web page that has the following HTML structure. (Fig 6)

Fig 6

```
<button id="submit-button" class="btn btn-primary">Submit</button>
```

Here's how you would write the Selenium IDE commands to find and click this button using different locator strategies. (Fig 7)

Fig 7

```
Command: click
Target: id=submit-button  // Using ID

Command: click
Target: css=button#submit-button  // Using CSS Selector

Command: click
Target: xpath=//button[text()='Submit']  // Using XPath
```

Testing for Button Presence

If you want to check if a button is present before clicking it, you can use the assertElementPresent command.

Example (Fig 8)

Fig 8

```
Command: assertElementPresent
Target: id=submit-button  // Check if the button is present
```

© NIMI NOT TO BE REPUBLISHED

Page source for debugging

Objective: At the end of this exercise you shall be able to
- **view page source to troubleshoot and debug issues.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Using the page source for debugging in Selenium IDE can help you understand the structure of the web page and troubleshoot issues in your automated tests. Here's a detailed guide on how to utilize page source effectively for debugging.

1 Accessing Page Source

To view the page source of a web page.

- **Browser Developer Tools**
  - Right-click on the web page and select **View Page Source** or press Ctrl + U (or Command + Option + U on macOS). This will open a new tab displaying the HTML source code of the page.
  - Alternatively, you can open the Developer Tools (F12 or Ctrl + Shift + I) and navigate to the Elements tab to see a live view of the DOM (Document Object Model).

2 Inspecting Elements

Using the Developer Tools, you can inspect specific elements on the page to understand their attributes, hierarchy, and styles.

- **Inspect Tool:** Click on the Select Element icon (a cursor over a square) in the Developer Tools and hover over the button or element you want to investigate. This will highlight the corresponding HTML in the Elements tab.
- Look for attributes like id, class, name, and type, which are useful for creating locators in your Selenium IDE test scripts.

3 Debugging with Page Source

When debugging test scripts, the page source can help you identify issues related to element locators or timing. Here are some common scenarios.

a Element not found

If your script fails because it cannot find an element, check the page source to ensure that

- The element exists on the page at the time the command is executed.
- The locator you are using matches the element's attributes.

Example (Fig 1)

Fig 1

```
Command: click
Target: id=submit-button  // Fails if the button does not exist or the ID is incorrect.
```

b Dynamic content

If the page content changes dynamically (e.g., through AJAX), ensure that the element is present when your command runs. You might need to add waits.

Example of Adding Wait (Fig 2)

Fig 2

```
Command: waitForElementPresent
Target: id=submit-button  // Wait for the button to appear in the DOM.
```

c Using Assertions

Utilize assertions to validate the presence and state of elements after actions are performed.

© NIMI NOT TO BE REPUBLISHED

Example (Fig 3)

Fig 3

```
Command: assertElementPresent
Target: id=submit-button  // Ensure the button is present after navigating to the page.
```

4 Viewing the Console Output

Sometimes errors may not be visible in the IDE. Check the browser's console for error messages (JavaScript errors, network issues, etc.) that could impact the page load or element rendering.

5 Debugging Example Using Page Source

Imagine a scenario where you have the following HTML for a login form. (Fig 4)

Fig 4

```
<form id="login-form">
    <input type="text" id="username" name="username" />
    <input type="password" id="password" name="password" />
    <button id="submit-button">Login</button>
</form>
```

Possible Debugging Steps

1 Check Element Availability

- After navigating to the login page, use the Developer Tools to confirm that the #login-form exists.

2 Run Selenium Commands (Fig 5)

Fig 5

```
Command: open
Target: http://example.com/login

Command: waitForElementPresent
Target: id=login-form  // Ensure the form is loaded

Command: type
Target: id=username
Value: testuser

Command: type
Target: id=password
Value: secretpassword

Command: click
Target: id=submit-button
```

3 Check for Errors

- If clicking the button results in an error, check the console for any JavaScript errors or network failures that might be affecting the form submission.

© NIMI NOT TO BE REPUBLISHED

Locator assistance

Objective: At the end of this exercise you shall be able to
- **help identify and use element locators effectively.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Locators are crucial in Selenium IDE as they are used to identify and interact with elements on a web page. Here's a comprehensive guide to understanding and using different types of locators effectively in Selenium IDE, along with tips for troubleshooting locator issues.

Types of Locators

1 ID Locator

- The id attribute is usually the most efficient and reliable way to locate an element since it should be unique within the page.
- **Example (Fig 1)**

Fig 1

```
Command: click
Target: id=submit-button
```

2 Name Locator

- The name attribute can also be used if it's unique for the element you're targeting.
- **Example (Fig 2)**

Fig 2

```
Command: type
Target: name=username
Value: testuser
```

3 CSS Selector

- CSS selectors provide a powerful way to select elements based on their attributes, classes, or positions.
- **Example**
  - By class (Fig 3)

Fig 3

```
Command: click
Target: css=.btn-primary  // Selects any element with the class 'btn-primary'
```

  - By element type and ID (Fig 4)

Fig 4

```
Command: click
Target: css=button#submit-button  // Selects a button with the ID 'submit-button'
```

4 XPath

- XPath is a versatile way to locate elements in the HTML structure. It can target elements based on their text, attributes, or hierarchy.
- **Examples**
  - By element text (Fig 5)

Fig 5

```
Command: click
Target: xpath=//button[text()='Submit']  // Clicks a button that has the text 'Subm
```

  - By attribute (Fig 6)

Fig 6

```
Command: click
Target: xpath=//input[@name='username']  // Selects an input with the name attribut
```

© NIMI NOT TO BE REPUBLISHED

5 Link Text and Partial Link Text

- If the element is a link, you can use the full text or part of the text to locate it.
- **Example (Fig 7)**

Fig 7

```
Command: click
Target: link=Click Here  // Clicks a link with the exact text 'Click Here'

Command: click
Target: partialLink=Click  // Clicks a link with partial text 'Click'
```

6 Class Name

- Use this if you want to select elements based on their class attribute.
- **Example (Fig 8)**

Fig 8

```
Command: click
Target: class=btn  // Selects an element with the class 'btn'
```

7 Tag Name

- This allows you to select elements based on their HTML tag (like <button>, <input>, etc.).
- **Example (Fig 9)**

Fig 9

```
Command: click
Target: tag=button  // Clicks the first button element found
```

Tips for Effective Locator Usage

1 Use Unique Locators

- Whenever possible, prefer locators that uniquely identify elements (like id or unique name attributes). This reduces ambiguity and increases reliability.

2 Combine Locators

- When elements share the same identifiers, you can combine locators using CSS selectors or XPath to be more specific.
- **Example (Fig 10)**

Fig 10

```
Command: click
Target: css=div#login .btn-primary  // Clicks a button with the class 'btn-primary'
```

3 Test Locators in the Browser

- Before using a locator in Selenium IDE, you can test it in the browser console. For example, you can use document.querySelector() for CSS selectors or document.evaluate() for XPath.

4 Inspect with Developer Tools

- Use the browser's Developer Tools (F12) to inspect elements. Right-click the element and select "Inspect" to view its attributes and try out different locators.

5 Check for Dynamic Elements

- If elements change based on user interaction (like AJAX-loaded content), ensure your locators are not dependent on dynamic values that might change during execution.

6 Handling Multiple Elements

- If a locator matches multiple elements, consider using indexes in your XPath or CSS selectors.
- **Example (Fig 11)**

Fig 11

```
Command: click
Target: xpath=(//button[@class='btn'])[1]  // Clicks the first button with the class
```

© NIMI NOT TO BE REPUBLISHED

Troubleshooting Locator Issues

1 Element Not Found

- Double-check the locator for typos or incorrect attributes.
- Ensure that the element is loaded and visible on the page before trying to interact with it.

2 Use Waits

- If elements are dynamically loaded, use wait commands to ensure the element is present before interacting.
- **Example (Fig 12)**

Fig 12

```
Command: waitForElementPresent
Target: id=submit-button
```

3 Consult the Page Source

- If locators fail, examine the page source to verify the element's existence and its attributes.

4 Try Different Locator Types

- If one locator type fails, try another. For example, switch from id to XPath or CSS if you encounter issues.

Example of Using Multiple Locators

Here's an example of a simple login test case using various locator strategies. (Fig 13)

Fig 13

```
Command: open
Target: http://example.com/login

Command: waitForElementPresent
Target: id=username  // Wait for the username field to be present

Command: type
Target: id=username
Value: testuser

Command: type
Target: id=password
Value: secretpassword

Command: click
Target: css=button[type='submit']  // Click the submit button using CSS Selector

Command: waitForElementPresent
Target: xpath=//h1[text()='Welcome']  // Wait for the welcome message to appear

Command: assertText
Target: xpath=//h1
Value: "Welcome"  // Validate that the user has been redirected to the welcome page
```

© NIMI NOT TO BE REPUBLISHED

Writing a test suite

Objective: At the end of this exercise you shall be able to
- **create a collection of related test cases for execution.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Creating a test suite in Selenium IDE is a powerful way to organize and manage your test cases. A test suite allows you to group related tests together, enabling you to run them as a single batch, which can streamline your testing process and ensure consistent execution. Below, I'll guide you through the steps to write and manage a test suite in Selenium IDE.

What is a Test Suite?

A test suite is a collection of test cases that are grouped together to test a specific functionality or feature of an application. Using test suites helps you run multiple tests simultaneously and manage them efficiently.

Steps to Write a Test Suite in Selenium IDE

1 Open Selenium IDE

- Launch the Selenium IDE application in your browser.

2 Create a New Project

- If you don't have a project yet, click on **File > New Project** to create one.
- Name your project appropriately (e.g., "UserLoginTests").

3 Create Test Cases

- **Create Individual Test Cases**
  - Click on the Test Cases tab.
  - Click on Add Test Case to create individual test cases.
  - Name your test cases based on their functionality (e.g., "ValidLogin", "InvalidLogin", "ForgotPassword").

- **Record or Write Test Steps**
  - You can either record interactions with the application or manually write the test steps for each case.
  - **Example of a Valid Login Test Case**

Fig 1

```
Command: open
Target: http://example.com/login

Command: type
Target: id=username
Value: validUser

Command: type
Target: id=password
Value: validPassword

Command: click
Target: css=button[type='submit']

Command: waitForElementPresent
Target: xpath=//h1[text()='Welcome']

Command: assertText
Target: xpath=//h1
Value: "Welcome"
```

4 Organize Test Cases into a Test Suite

- After creating your test cases, navigate to the Test Suites tab.
- Click on Add Test Suite to create a new suite.
- Name your test suite (e.g., "LoginTestSuite").
- You will see a list of your test cases on the left side.

5 Add Test Cases to the Suite

- Select the test cases you want to include in the suite and use the Add button to include them in your suite.
- You can reorder the test cases within the suite to specify the order of execution.

© NIMI NOT TO BE REPUBLISHED

6 Configure Test Suite Settings (Optional)

- You can set options for the test suite, such as Run Mode (sequential or parallel execution) or Test Suite Variables that can be used across test cases.

7 Save Your Project

- Make sure to save your project frequently to avoid losing your work. Click on **File > Save Project.**

8 Run the Test Suite

- Click on the Run button next to your test suite to execute all the included test cases.
- You can view the results and logs in the Selenium IDE's output window to check for any failures or errors.

Example of a Test Suite Structure

Here's an example of how your test suite may look:

Test Suite: LoginTestSuite

- **Test Case 1: ValidLogin**
  - Steps to perform a valid login and verify the welcome message.
- **Test Case 2: InvalidLogin**
  - Steps to attempt a login with invalid credentials and verify the error message.
- **Test Case 3: ForgotPassword**
  - Steps to test the forgot password functionality and verify the reset email is sent.

© NIMI
NOT TO BE REPUBLISHED

User extensions

Objective: At the end of this exercise you shall be able to
- **custom functions or libraries to extend selenium capabilities.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

In Selenium IDE, user extensions are a powerful way to enhance the functionality of your test scripts by allowing you to write custom commands, add additional logic, or integrate external libraries. User extensions can help you perform tasks that may not be supported out of the box in Selenium IDE, such as complex assertions, custom waits, or API interactions.

How to Use User Extensions in Selenium IDE

1 Understanding User Extensions

User extensions in Selenium IDE are JavaScript files that define custom commands or augment the existing capabilities of Selenium IDE. These extensions allow for greater flexibility and can be tailored to meet specific testing needs.

2 Creating a User Extension

To create a user extension, follow these steps:

- **Create a JavaScript File**
  - Write a JavaScript file that defines the custom commands you want to add.
  - Save it with a .js extension (e.g., customCommands.js).
- **Define Custom Commands:** Here's an example of how to create a simple custom command in your JavaScript file. (Fig 1)

Fig 1

```
// customCommands.js

// Custom command to wait for an element to be visible
Selenium.prototype.doWaitForVisible = function(target) {
    const element = this.page().findElement(target);
    return this.waitForCondition(`window.getComputedStyle(document.querySelector('${target}'
};
```

- This custom command waits for an element to become visible before continuing with the test.

3 Loading User Extensions in Selenium IDE

After you've created your user extension

- **Open Selenium IDE:** Launch the Selenium IDE application.
- **Access the Options Menu:** Click on the Options button (gear icon) in the top right corner.
- **Select User Extensions:** In the options menu, find the User Extensions section.
- **Add Your Extension:** Click on Add and select the JavaScript file you created (e.g., customCommands.js). This will load the custom commands into Selenium IDE.

4 Using Custom Commands in Your Tests

Once your user extension is loaded, you can use the custom commands in your test cases just like built-in commands.

Example (Fig 2)

Fig 2

```
Command: doWaitForVisible
Target: css=.my-element  // Use the custom command to wait for an element to be visible
```

© NIMI NOT TO BE REPUBLISHED

Example of a Complete User Extension

Here's a more comprehensive example of a user extension that adds multiple custom commands. (Fig 3)

Fig 3

```
// customCommands.js

// Custom command to wait for an element to be visible
Selenium.prototype.doWaitForVisible = function(target) {
    const element = this.page().findElement(target);
    return this.waitForCondition(`window.getComputedStyle(document.querySelector('${target}')).c
};

// Custom command to click an element with a retry mechanism
Selenium.prototype.doClickWithRetry = function(target, retries) {
    const command = () => {
        this.click(target);
    };
    return this.retry(command, retries || 3);
};

// Custom command to verify an element's text
Selenium.prototype.doVerifyText = function(target, expectedText) {
    const actualText = this.getText(target);
    if (actualText !== expectedText) {
        throw new Error(`Expected text: "${expectedText}", but found: "${actualText}"`);
    }
};
```

Tips for Creating User Extensions

- **Keep It Simple:** Start with simple commands before adding complexity. This makes debugging easier.
- **Test Thoroughly:** Ensure that your custom commands work as expected by testing them within Selenium IDE.
- **Use Descriptive Names:** Name your custom commands descriptively to indicate their purpose and functionality.
- **Leverage Existing Libraries:** If you need more advanced functionality, consider leveraging existing JavaScript libraries (like jQuery) within your extensions.

Troubleshooting User Extensions

- **Check Console for Errors:** If your custom command isn't working, open the console in your browser's Developer Tools to check for errors in your JavaScript.
- **Ensure Proper Loading:** Make sure that your user extension file is correctly loaded in Selenium IDE.
- **Debug Commands:** Use console.log statements in your custom commands to trace execution flow and variable values.

© NIMI NOT TO BE REPUBLISHED

Software Testing Assistant - Web Automated Testing Tool & Comparison

Format

Objective: At the end of this exercise you shall be able to
- **define structure or layout for data or code organization.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Creating a user extension in Selenium IDE involves writing JavaScript to define custom commands and integrating it into your test automation workflow. Here's a structured format to guide you through the process:

User Extension Format for Selenium IDE

1 Purpose

- Define the goal of your user extension. Specify what functionality or commands you want to add to Selenium IDE.

2 Create a JavaScript File

- Save your JavaScript code in a file with a .js extension (e.g., customCommands.js).

3 Define Custom Commands

- Structure your JavaScript code to define new commands clearly. Here's a template. (Fig 1)

Fig 1

```
// Custom command to wait for an element to be visible
Selenium.prototype.doWaitForVisible = function(target) {
    const element = this.page().findElement(target);
    return this.waitForCondition(
        `window.getComputedStyle(document.querySelector('${target}')).display !== 'none'`,
        5000
    );
};

// Custom command to click an element with a retry mechanism
Selenium.prototype.doClickWithRetry = function(target, retries) {
    const command = () => {
        this.click(target);
    };
    return this.retry(command, retries || 3);
};

// Custom command to verify an element's text
Selenium.prototype.doVerifyText = function(target, expectedText) {
    const actualText = this.getText(target);
    if (actualText !== expectedText) {
        throw new Error(`Expected text: "${expectedText}", but found: "${actualText}"`);
    }
};
```

© NIMI NOT TO BE REPUBLISHED

4 Loading User Extensions in Selenium IDE

- Open Selenium IDE
- Access Options
  - Click on the Options button (gear icon).
- Select User Extensions
  - Go to the User Extensions section.
- Add Your Extension
  - Click Add and select your .js file.

5 Using Custom Commands in Tests

- After loading, use your custom commands in test cases like any built-in command. (Fig 2)

Fig 2

```
Command: doWaitForVisible
Target: css=.my-element  // Waits for an element to be visible

Command: doClickWithRetry
Target: id=retry-button  // Clicks the button with retry logic

Command: doVerifyText
Target: xpath=//h1
Value: "Expected Title"  // Verifies the text of an element
```

6 Best Practices

- **Keep Commands Simple:** Start with straightforward functionality.
- **Test Thoroughly:** Run your custom commands in Selenium IDE to ensure they work correctly.
- **Use Descriptive Names:** Name commands intuitively to convey their functionality.
- **Leverage Libraries:** Use libraries like jQuery if more advanced features are needed.

7 Debugging Tips

- **Console Logging:** Use console.log to debug command execution.
- **Check for Errors:** Use the browser's Developer Tools to look for any JavaScript errors in the console.
- **Verify File Loading:** Ensure your .js file is properly loaded in Selenium IDE.

8 Example Commands (Fig 3)

Fig 3

```
// Example of additional custom commands
Selenium.prototype.doHover = function(target) {
    const element = this.page().findElement(target);
    const event = new MouseEvent('mouseover', { bubbles: true });
    element.dispatchEvent(event);
};

Selenium.prototype.doCaptureScreenshot = function() {
    // Logic to capture a screenshot
    console.log("Screenshot captured!");
};
```

© NIMI NOT TO BE REPUBLISHED

Executing selenium - IDE tests on different browsers

Objective: At the end of this exercise you shall be able to
- **run selenium-IDE tests across multiple browsers for cross-browser compatibility.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Executing Selenium IDE tests on different browsers allows you to ensure that your web application functions correctly across various environments. Selenium IDE supports multiple browsers, and you can run your tests in them with minimal configuration. Here's how you can set up and execute Selenium IDE tests on different browsers.

Supported Browsers

Selenium IDE typically supports the following browsers:

- **Google Chrome**
- **Mozilla Firefox**
- **Microsoft Edge**
- **Safari** (limited support)

Steps to Execute Selenium IDE Tests on Different Browsers

1 Install Selenium IDE

- Ensure that you have the Selenium IDE installed in your preferred browser. You can find it in the Chrome Web Store or Firefox Add-ons.

2 Create Your Test Cases

- Open Selenium IDE and create your test cases as needed. Make sure they are working as expected in your default browser.

3 Export Test Cases (Optional)

- If you need to run the tests outside of Selenium IDE (for example, in a different environment or using Selenium WebDriver), you can export your test cases to a format compatible with other testing frameworks (like Java, Python, etc.).
- To export
  - Select the test case or suite you want to export.
  - Click on File > Export and choose the format (e.g., Java, Python).

4 Open the Desired Browser

- Launch the browser in which you want to execute the tests (e.g., Chrome, Firefox, Edge).

5 Run Tests in the Selected Browser

- **For Selenium IDE**
  - Open Selenium IDE in the browser where you want to run the tests.
  - Load your project (if not already loaded).
  - Click on the Run button for the test case or suite you want to execute.
- **Switching Browsers**
  - If you want to switch to a different browser, you will need to open the Selenium IDE in that browser and run the tests again.

6 Cross-Browser Execution Using Selenium WebDriver

- If you want to execute your tests in multiple browsers automatically, you might consider using Selenium WebDriver with a testing framework (like TestNG, JUnit, or Pytest).
- **Example Setup:**
  - Export your tests from Selenium IDE.
  - Set up a WebDriver for each browser in your test script.
  - Run the tests using a loop or parallel execution.

7 Use Docker or Selenium Grid (Optional)

- For more extensive testing across multiple browsers and versions, consider using Selenium Grid or Docker containers with browser images.
- Selenium Grid allows you to run tests on multiple machines and browsers simultaneously.
- Docker provides a lightweight way to run different browser environments.

© NIMI NOT TO BE REPUBLISHED

Example of Running Tests with Selenium WebDriver

Here's an example of how you might set up a simple cross-browser test in Java using Selenium WebDriver:

Fig 1

```java
public class CrossBrowserTest {
    public static void main(String[] args) {
        String browser = "chrome"; // Change to "firefox" or "edge" to test other browsers

        WebDriver driver;

        switch (browser.toLowerCase()) {
            case "chrome":
                System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
                driver = new ChromeDriver();
                break;
            case "firefox":
                System.setProperty("webdriver.gecko.driver", "path/to/geckodriver");
                driver = new FirefoxDriver();
                break;
            case "edge":
                System.setProperty("webdriver.edge.driver", "path/to/edgedriver");
                driver = new EdgeDriver();
                break;
            default:
                throw new IllegalArgumentException("Browser not supported: " + browser);
        }

        driver.get("http://example.com");

        // Add your test logic here

        driver.quit();
    }
}
```

© NIMI NOT TO BE REPUBLISHED

Best Practices for Cross-Browser Testing

- **Consistent Test Data:** Ensure that you use the same test data across different browsers to maintain consistency in results.
- **Handle Browser-Specific Behaviors:** Some browsers may handle elements or scripts differently; be prepared to implement workarounds if needed.
- **Use Browser Developer Tools:** Utilize the developer tools in each browser to inspect elements and troubleshoot issues during testing.
- **Regularly Update Drivers:** Keep your WebDriver executables updated to ensure compatibility with the latest browser versions.

Troubleshooting

Objective: At the end of this exercise you shall be able to
- **identify and resolve issues during test execution.**

Requirements

Tools/Equipment/Machines

- A working PC with IDE Software - 1 No.

PROCEDURE

Troubleshooting in Selenium IDE – Step-by-Step Guide with Output

Troubleshooting in **Selenium IDE** helps resolve issues in test automation, such as element locators not working, timing issues, or incorrect test logic. Selenium IDE provides tools like the **Log Panel, Breakpoints, Step Execution, and Assertions** to debug test cases effectively.

Step-by-Step Troubleshooting Guide

1 Open Selenium IDE and Run Your Test Case

- Open Selenium IDE in your browser.
- Load or record a test case.
- Click "Run Current Test Case" (▶) to execute the script.
- If an error occurs, check the Log Panel.

2 Check the Log Panel for Errors

The Log Panel at the bottom of Selenium IDE displays execution results and errors.

Example Error Output (Fig 1)

Fig 1

```
[ERROR] Element not found: id=loginButton
```

Fix: Check if the locator (ID, XPath, CSS Selector) is correct.

- Use the "Find" button in Selenium IDE to validate the locator.
- Inspect the element in the browser using DevTools (F12 → Elements tab).

3 Use Breakpoints to Pause Execution

Breakpoints allow you to pause execution at a specific step and inspect the test state.

How to Add a Breakpoint

- Right-click on a command.
- Select "Toggle Breakpoint".
- A red dot appears next to the command.

Example Test Case with Breakpoint (Fig 2)

Fig 2

Command	Target	Value
`open`	`https://example.com`	
`click`	`id=loginButton`	
`storeText`	`id=welcomeText`	`message`

Expected Output in Log Panel (Fig 3)

Fig 3

```
[INFO] Paused at command: click | id=loginButton
```

Fix

- Use Step Over (■) to execute the next step manually.
- Check if the button exists before the click command.

4 Check Variables Panel for Stored Values

The Variables Panel stores values from the test script. If a value is missing or incorrect, the test might fail.

© NIMI NOT TO BE REPUBLISHED

Example (Fig 4)

Fig 4

Command	Target	Value
storeText	id=welcomeText	message

If welcomeText contains "Welcome, John!", the Log Panel will show (Fig 5)

Fig 5

```
[INFO] Stored message = Welcome, John!
```

Fix: If message is empty, check if:

- The element id=welcomeText is correct.
- The page fully loads before retrieving text.

5 Handle Timing Issues (Use waitFor Commands)

If Selenium IDE tries to interact with an element before it's visible, use waitFor commands.

Example Fix – Add a Wait Before Clicking (Fig 6)

Fig 6

Command	Target	Value
waitForElementVisible	id=loginButton	5000
click	id=loginButton	

Fix: Ensures the button is visible before clicking.

6 Validate Assertions (Verify Correct Page State)

Use assert or verify commands to check page content.

Example: Verify Login Success Message (Fig 7)

Fig 7

Command	Target	Value
click	id=loginButton	
verifyText	id=welcomeText	Welcome, John!

Fix: If the assertion fails

- Verify if the login process works correctly.
- Check if the correct page is loaded.

7 Restart Test Execution (Clear Cache & Refresh)

If the issue persists, try

1 Clearing browser cache and cookies.
2 Restarting Selenium IDE.
3 Re-running the test after debugging steps.

© NIMI NOT TO BE REPUBLISHED