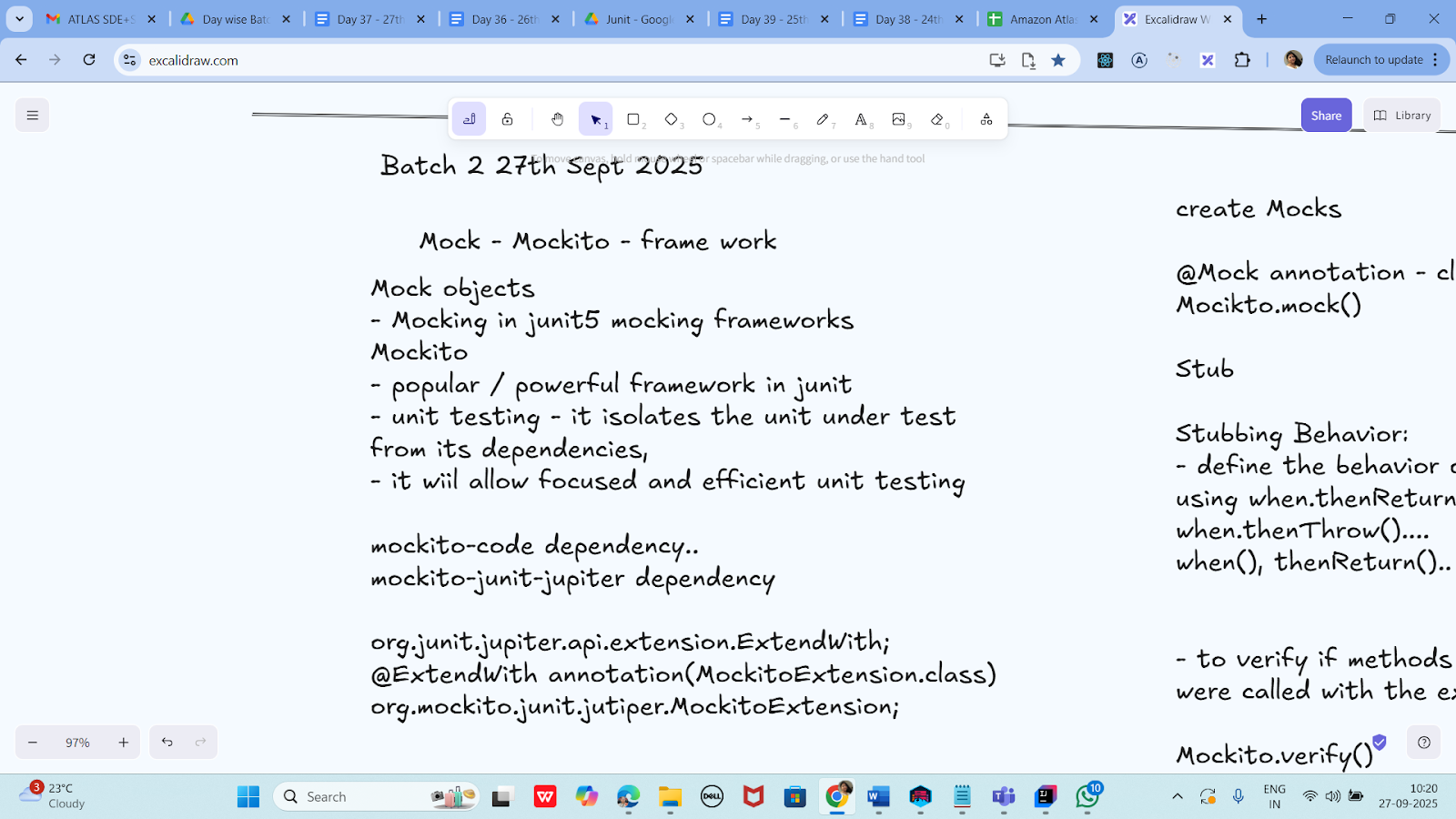
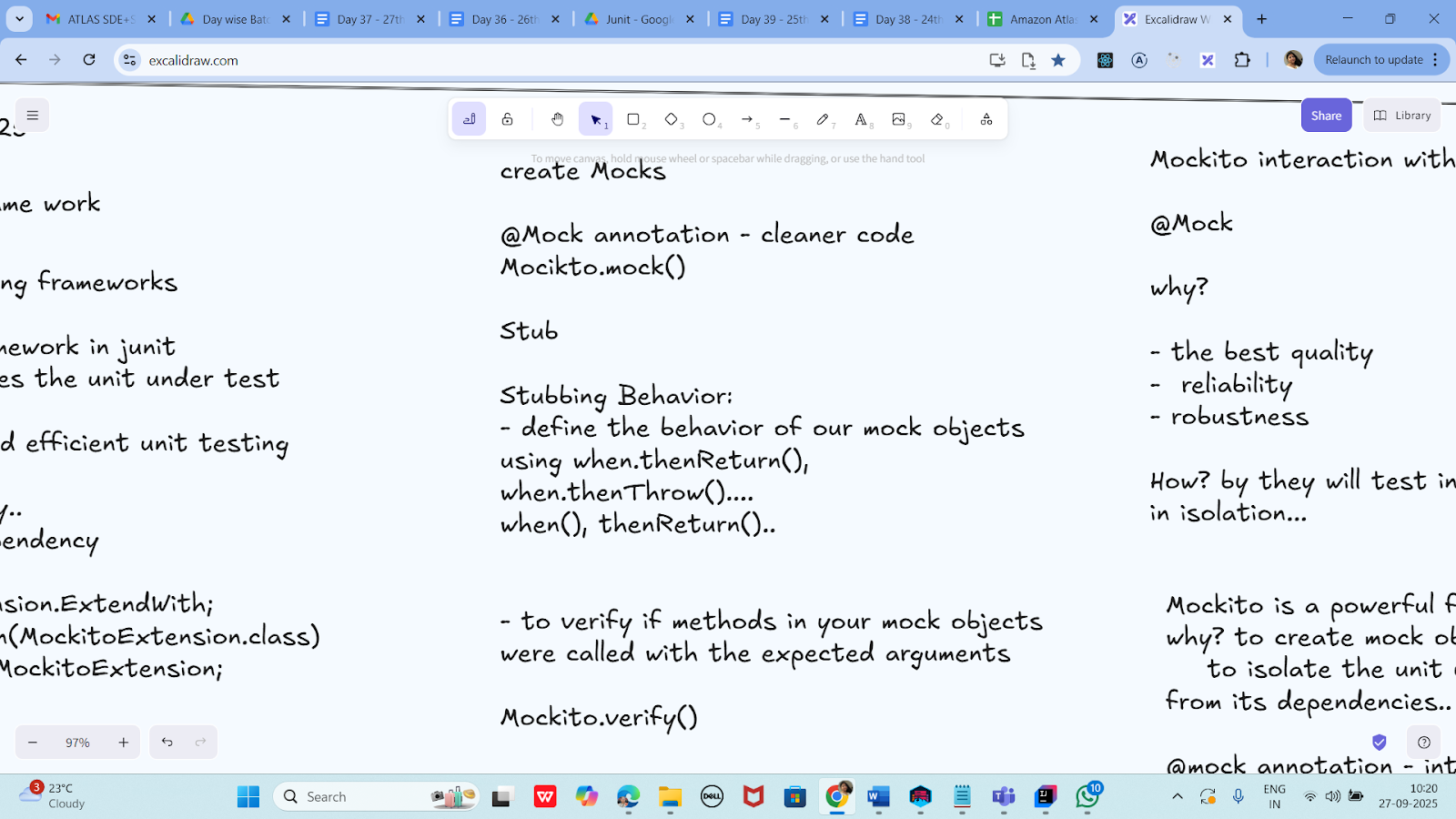
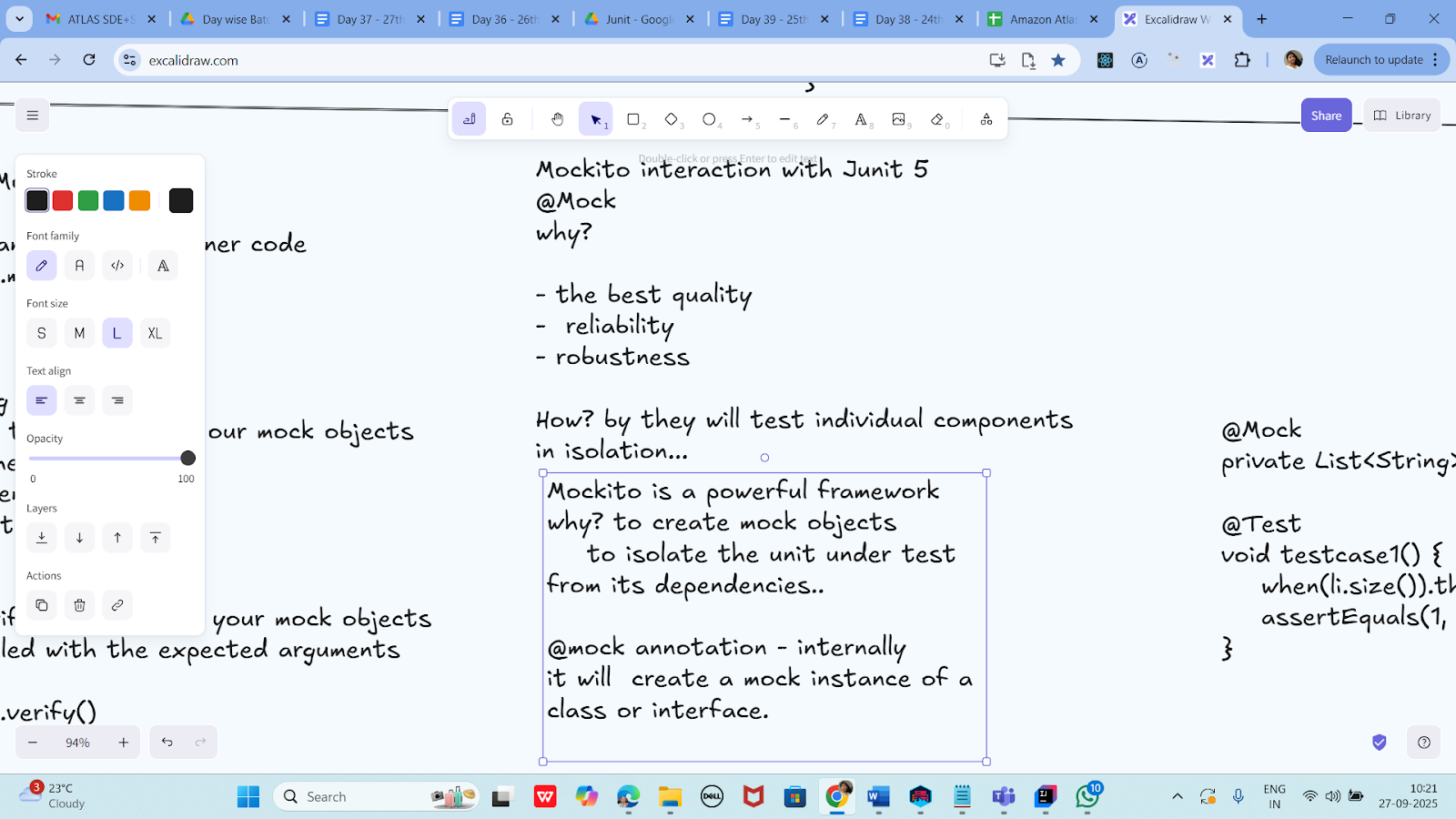
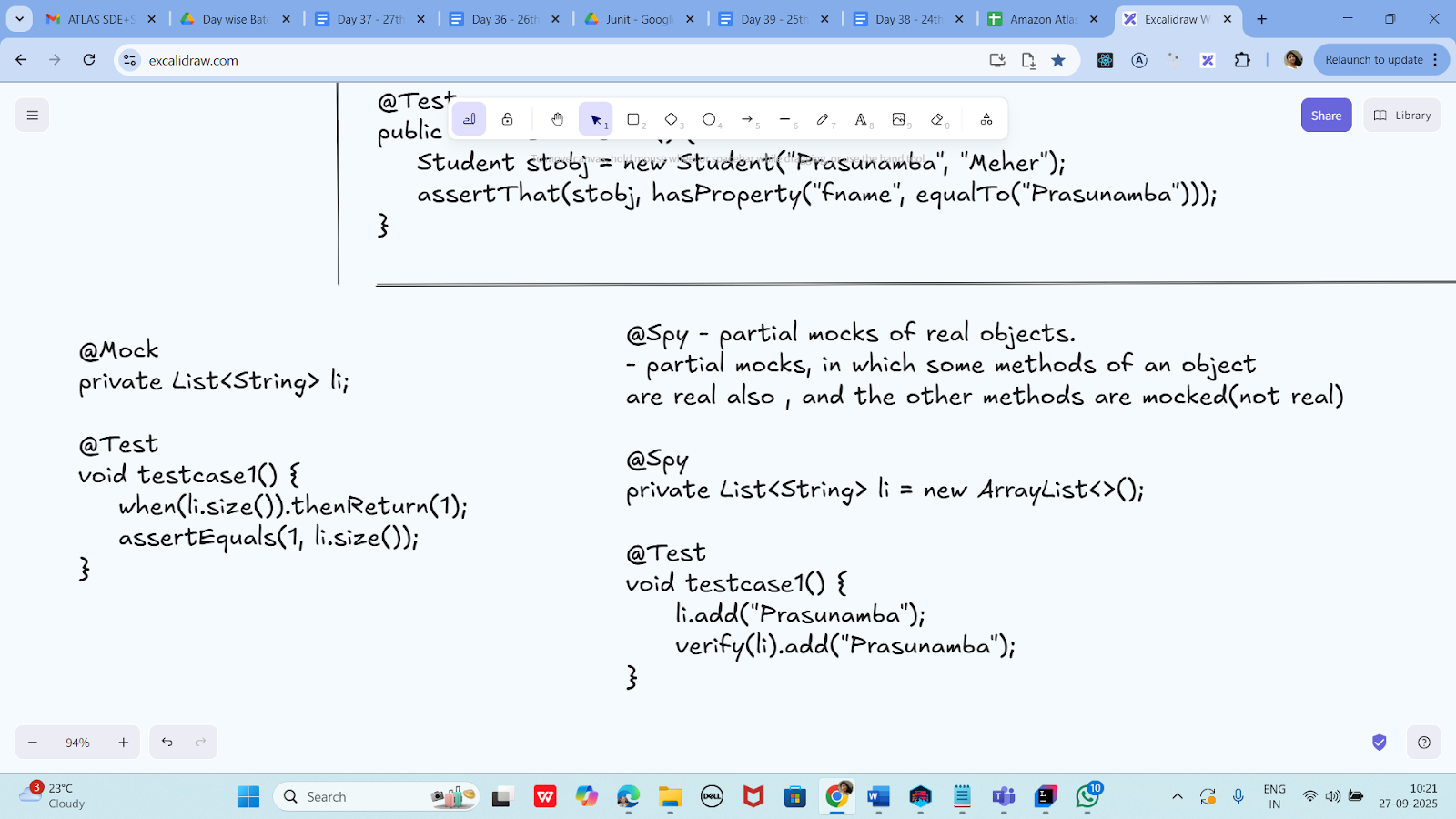
Day 37 - 27th Sept 2025

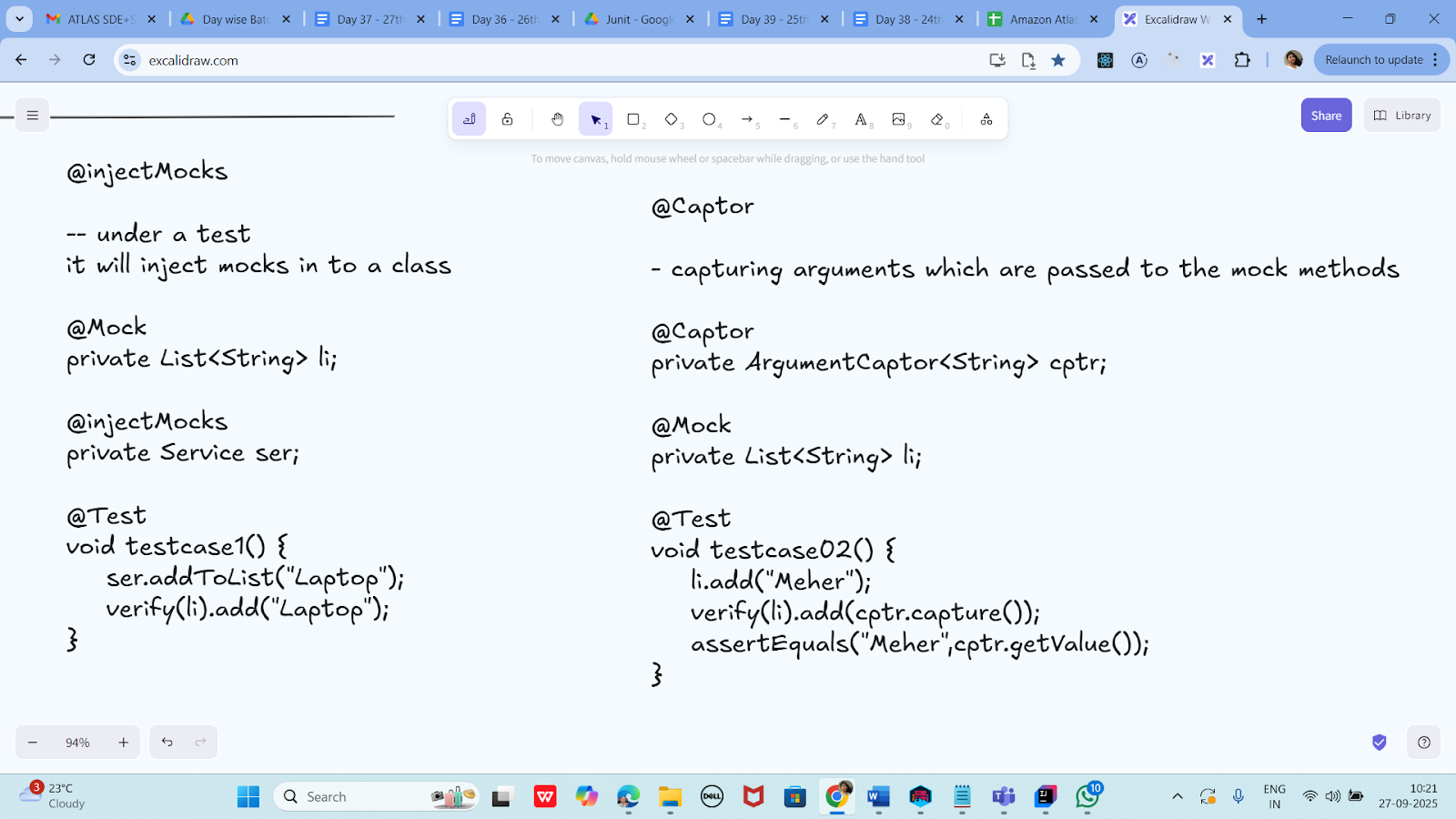
Mock objects



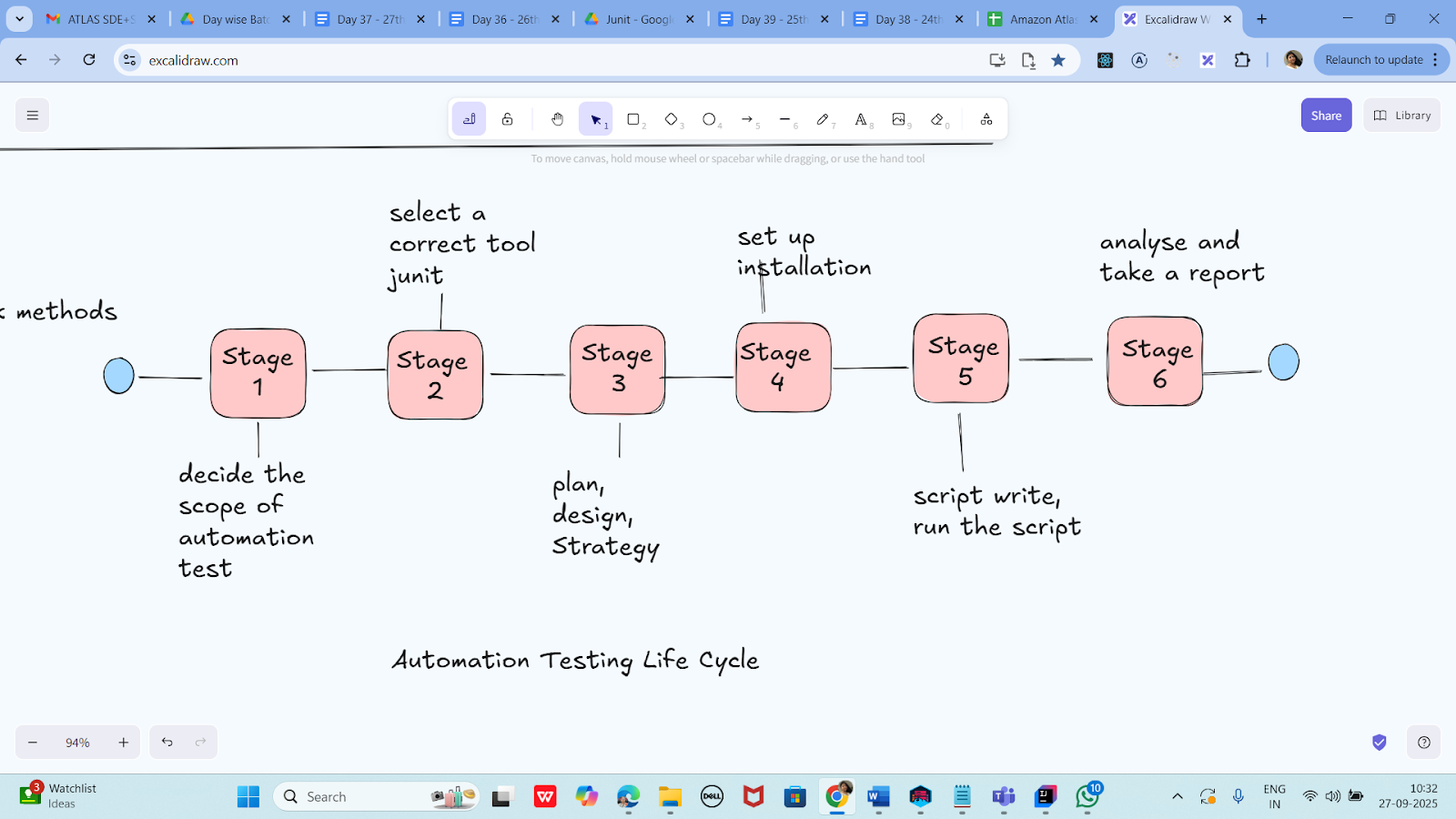


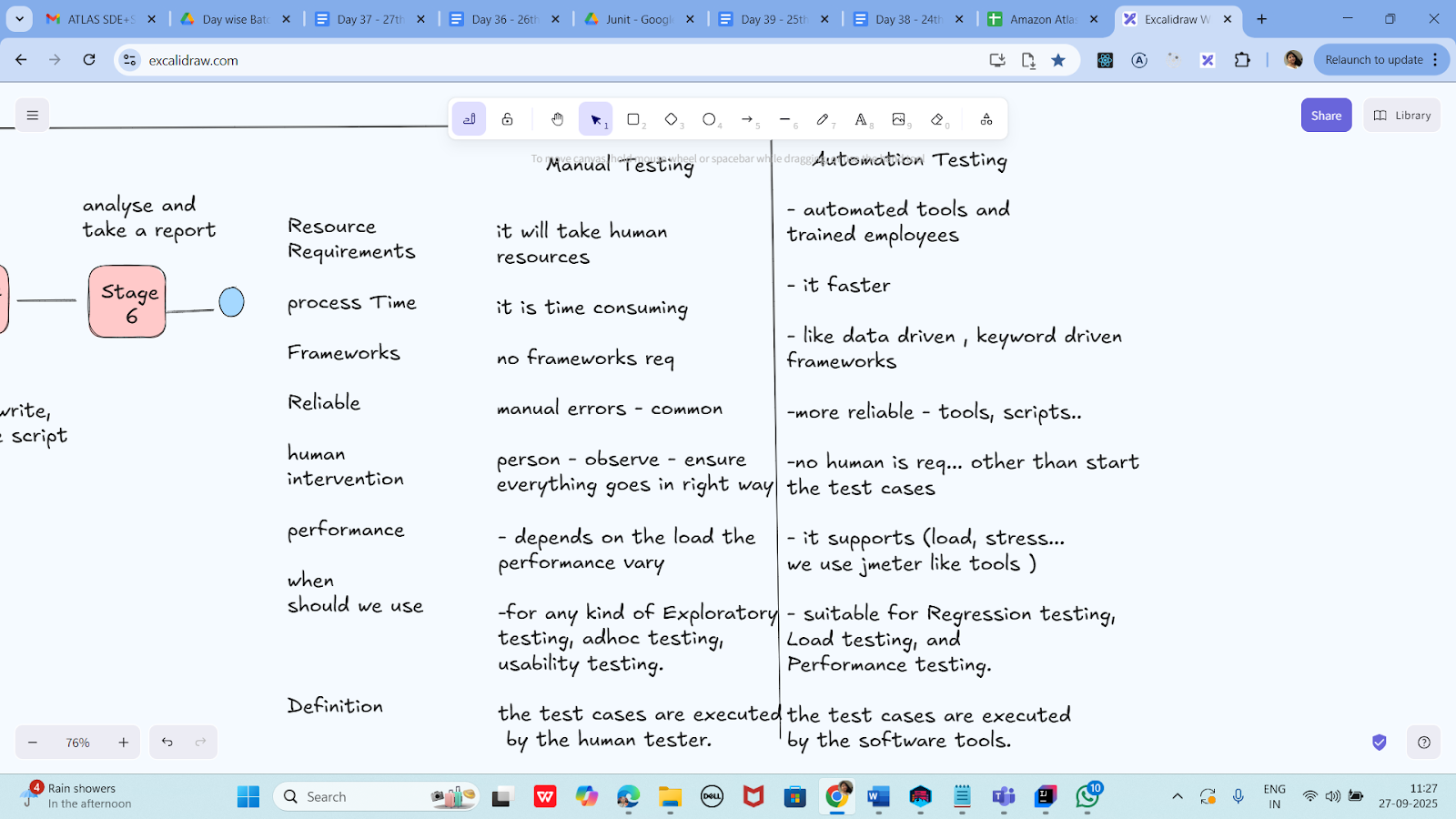






ATLC - automation Testing life Cycle





Task 01:

Mock Practice:

<groupId>org.mockito</groupId>

<artifactId>mockito-junit-jupiter</artifactId>

<version>5.12.0</version> <!-- Use the latest stable version -->

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.12.0</version>

<scope>test</scope>

</dependency>

User service .java

package org.example;

import java.util.Date;

public class UserService {

private final UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

public User02 registerUser(User02 user) {

user.setRegistrationDate(new Date());

return userRepository.save(user); //.save(user);

}

}

User repo.java

package org.example;

public interface UserRepository {

User02 findById(Long id);

User02 save(User02 user);

}

package org.example;

import java.util.Date;

public class User02 {

private Long id;

private String name;

private Date registrationDate;

// Getters and setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Date getRegistrationDate() {

return registrationDate;

}

public void setRegistrationDate(Date registrationDate) {

this.registrationDate = registrationDate;

}

}

package org.example;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Date;

import static org.junit.jupiter.api.Assertions.*assertNotNull*;

import static org.mockito.ArgumentMatchers.*any*;

import static org.mockito.Mockito.*when*;

@ExtendWith(MockitoExtension.class)

public class DemoTest07Mockito {

@Mock

private UserRepository userRepository;

@InjectMocks

private UserService userService;

@Test

void testRegisterUser() {

User02 newUser = new User02();

newUser.setName("Pratheesh");

*when*(userRepository.save(*any*(User02.class))).thenReturn(newUser);

User02 registeredUser = userService.registerUser(newUser);

*assertNotNull*(registeredUser);

*assertNotNull*(registeredUser.getRegistrationDate());

}

}

Task 02:

package org.example;

public class ToyService {

// Returns the toy name for a given toy id

public String getToyName(int toyId) {

switch (toyId) {

case 1:

return "Lego";

case 2:

return "Barbie";

default:

// Fallback if unknown toy

return getFallbackName();

}

}

// Method used as fallback in the spy test

public String getFallbackName() {

return "Unknown toy";

}

}

package org.example;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.Spy;

import org.mockito.junit.jupiter.MockitoExtension;

import static org.junit.jupiter.api.Assertions.*assertEquals*;

import static org.mockito.Mockito.\*;

@ExtendWith(MockitoExtension.class)

class DemoTest08Mockito {

@Spy

ToyService spyToyService;

@Test

void method1() {

// When

String name = spyToyService.getToyName(1);

// Then

*assertEquals*("Lego", name);

// We can still verify interactions with the spy

*verify*(spyToyService, *times*(1)).getToyName(1);

}

@Test

void method2() {

*doReturn*("Default toy").when(spyToyService).getFallbackName();

// When

String name = spyToyService.getToyName(3);

// Then

*assertEquals*("Default toy", name);

// Verify that getFallbackTitle was called exactly once

*verify*(spyToyService, *times*(1)).getFallbackName();

}

}

Task 03:

Spying a service

package org.example;

public class BookRepository {

public int getBookCount() {

System.*out*.println("assume calling real getBookCount() from the database.");

return 10;

}

}

package org.example;

public class BookService {

private final BookRepository bookRepository;

public BookService(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public boolean isBookCountLow() {

int count = bookRepository.getBookCount();

System.*out*.println("Real method isBookCountLow() called.");

return count < 10;

}

public String getBookServiceStatus() {

if (isBookCountLow()) {

return "LOW\_STOCK";

}

return "IN\_STOCK";

}

}

package org.example;

//spying BookService

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.Spy;

import org.mockito.junit.jupiter.MockitoExtension;

import static org.junit.jupiter.api.Assertions.*assertEquals*;

import static org.mockito.Mockito.\*;

@ExtendWith(MockitoExtension.class)

class DemoTest09Mockito {

@Mock

BookRepository bookRepository;

@Spy

@InjectMocks

BookService bookServiceSpy;

@Test

void shouldReturnInStockWhenCountIsNormal() {

*when*(bookRepository.getBookCount()).thenReturn(20);

String status = bookServiceSpy.getBookServiceStatus();

*verify*(bookServiceSpy, *times*(1)).isBookCountLow();

*assertEquals*("IN\_STOCK", status);

}

@Test

void shouldReturnLowStockWhenIsBookCountLowIsStubbed() {

*doReturn*(true).when(bookServiceSpy).isBookCountLow();

String status = bookServiceSpy.getBookServiceStatus();

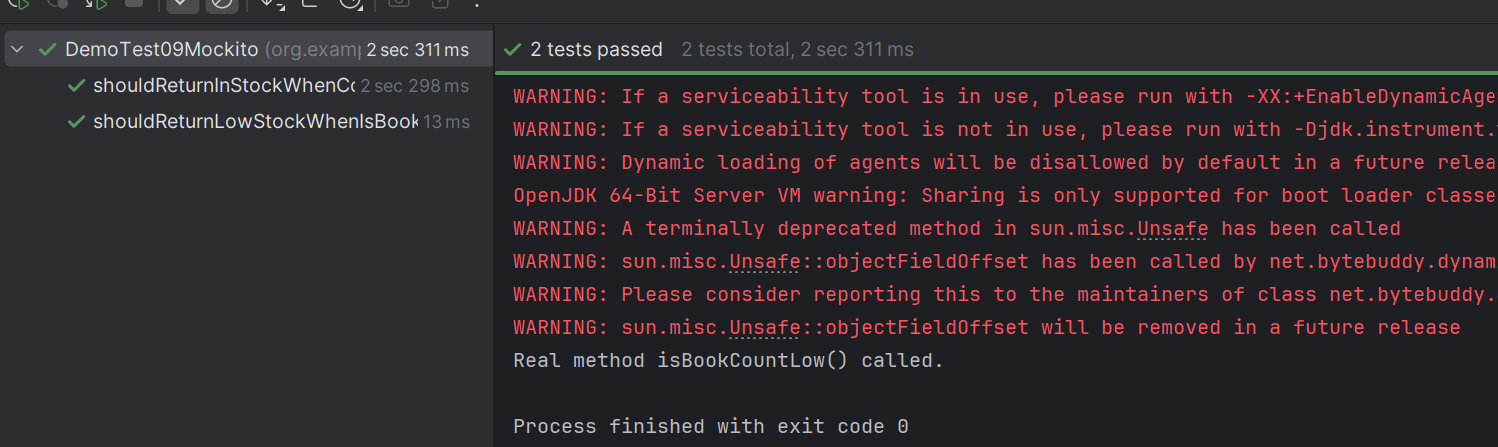
*verify*(bookServiceSpy, *times*(1)).isBookCountLow();

*assertEquals*("LOW\_STOCK", status);

*verify*(bookRepository, *never*()).getBookCount();

}

}



Task 04 to Task 08:

Hope you have gone through all the yesterdays tasks

Write Test cases for mobile app

Task 09:

In the context of Automation Testing, how does the Page Object Model (POM) improve test maintainability?

1. POM creates separate XML files for each UI element which reduces hardcoding in test scripts.

2. POM introduces a single test method for all page elements, reducing redundancy.

3. POM stores test data in property files which helps in data-driven testing.

4. POM abstracts UI elements as objects in separate classes, which isolates changes in the UI from the test logic.

Task 10:

In a Page Object Model (POM) framework, where should test data ideally reside?

1. In the page class itself for better cohesion.

2. In the test script, embedded as literals for simplicity.

3. In external files such as JSON, Excel, or property files to separate logic from data.

4. In browser cookies to simulate user sessions.

Task 11:

A team is building a Page Object Model (POM) automation framework. However, they observe test failures due to frequent changes in UI locators across environments. What should they focus on improving in their automation approach?

1. Use absolute XPath locators as they are more stable across environments.

2. Store locators in external property files and abstract them within page classes to isolate UI changes.

3. Implement a retry mechanism in every step to counter locator issues.

4. all locators to dynamic XPath expressions at runtime.

Task 12:

Why are mock objects commonly used in JUnit tests for services that call external APIs?

1. They simulate external APIs and allow testing of service logic in isolation without depending on actual API availability or response time.

2. They provide GUI forms for mocking user input.

3. They automatically generate alternate test scenarios.

4. They allow browser-based execution for all test cases.

Task 13:

You’ve written a JUnit test for an API that sends email notifications. Running the test actually sends emails, causing issues. How should this be addressed?

1. Replace the test logic with a mock that simulates email service behavior.

2. Remove the test as it affects production systems.

3. Use assertNull() on email object to prevent sending.

4. Allow emails but send them to a dummy address.

Task 14:

Which of the following best describes the role of test suites in JUnit?

1. Test suites are used to group test methods within a single class for reporting.

2. Test suites allow grouping multiple test classes and running them together, often used for regression or integration testing.

3. Test suites run a single test class multiple times with different parameters.

4. Test suites are used to auto-generate mock objects for integration with third-party services.

Task 15:

Why are mock objects used in JUnit testing, especially in enterprise applications?

1. They eliminate the need for assertions by capturing logs during execution.

2. They replace real dependencies like databases or web services to test components in isolation and ensure reliability.

3. They automate UI validations by replicating the behavior of actual user interactions.

4. They allow for runtime generation of test data based on machine learning techniques.

Task 16:

What does the annotation @Disabled do in a JUnit test?

1. It is used to generate random input values for the test cases.

2. It ensures that the test case will only run if another test fails.

3. It replaces the test method with a mock version automatically during runtime.

4. It marks a test case to be skipped during execution without deleting or commenting out the code.

Task 17:

What is the primary purpose of using assertions in JUnit tests?

1. Assertions are used to execute code blocks before and after the test cases for logging purposes.

2. Assertions ensure that certain exceptions are thrown during the execution of test methods.

3. Assertions define expected outcomes in tests and help automatically verify the correctness of code behavior.

4. Assertions are used to configure the order of test execution in test suites.

Task 18:

What is one of the key differences between manual testing and automation testing from a scalability perspective?

1. Manual testing scales better because it allows human intuition in exploratory testing scenarios.

2. Automation testing is ideal for scaling regression tests across builds and environments as scripts can run unattended.

3. Manual testing uses more reliable tools compared to automation which may fail silently.

4. Automation testing allows for ad-hoc testing with high flexibility in test case creation.

Task 19:

What is the purpose of the assertThat() method in conjunction with the Hamcrest library in JUnit?

1. It improves performance by running multiple assertions in parallel threads.

2. It allows developers to create GUI-based test results with better formatting.

3. It provides a more flexible and readable way to define expectations using matchers like containsString, hasItems, etc.

4. It is primarily used to automate exception handling by wrapping assertions in try-catch blocks.

Task 20:

How does the concept of timeout help improve the reliability of test execution in JUnit?

1. Timeout ensures that tests always return true after a specified duration, indicating success.

2. Timeout ignores any assertion failures if a method takes too long to execute.

3. Timeout helps identify tests that hang or take unusually long to execute, allowing early detection of performance issues.

4.Timeout marks the test as passed even if it is interrupted by a system signal or thread abort

Info Box:

<https://excalidraw.com/#json=r1ikvqPdKwwPMxI6IdEn2,ckA275OQLq5itYZ98c0FxQ>

Excalidraw updated at 11.30