Test Automation Framework - Best Practices Cheatsheet

1. KISS Principle (Keep It Simple)

Break complex tests into smaller, reusable modules. Avoid over-engineering.

```
public WebDriver initBrowser() {
    return new ChromeDriver(); // No static for parallel tests
}

@Test
public void testHomePageTitle() {
    WebDriver driver = initBrowser();
    driver.get("https://example.com");
    Assert.assertEquals("Expected Title", driver.getTitle());
    driver.quit();
}
```

2. Modular Approach

Separate test data, utilities, page objects, and test execution into modules.

```
public class LoginPage {
    @FindBy(id = "username") private WebElement usernameField;
    @FindBy(id = "password") private WebElement passwordField;
    @FindBy(id = "login") private WebElement loginButton;

    public LoginPage(WebDriver driver) {
        PageFactory.initElements(driver, this);
    }

    public void login(String username, String password) {
        usernameField.sendKeys(username);
        passwordField.sendKeys(password);
        loginButton.click();
    }
}
```

3. Setup Data via API/DB

Use APIs or database calls for test preconditions instead of UI.

```
public static void createUser(String username, String email) {
    Response response = RestAssured.given()
        .contentType("application/json")
        .body("{\"username\":\"" + username + "\", \"email\":\"" + email + "\"}")
        .post("https://api.example.com/users");
    Assert.assertEquals(201, response.getStatusCode());
}
```

4. Avoid Excel for Test Data

Use JSON, XML, CSV, or databases instead of Excel files.

```
public static UserData getUserData(String filePath) throws Exception {
    ObjectMapper mapper = new ObjectMapper();
    return mapper.readValue(new File(filePath), UserData.class);
}

public class UserData {
    private String username;
    private String email;
    // getters and setters
}
```

5. Design Patterns

Factory Pattern

Create objects based on input parameters.

```
public class DriverFactory {
    public static Driver getDriver(String browserType) {
        if (browserType.equalsIgnoreCase("chrome")) {
            return new ChromeDriverImpl();
        } else if (browserType.equalsIgnoreCase("firefox")) {
            return new FirefoxDriverImpl();
        }
        throw new IllegalArgumentException("Invalid browser type");
    }
}
```

Strategy Pattern

Switch between different algorithms or implementations.

```
public class BrowserContext {
    private BrowserStrategy strategy;

    public BrowserContext(BrowserStrategy strategy) {
```

```
        this.strategy = strategy;
    }

    public WebDriver executeStrategy() {
        return strategy.setUp();
    }
}
```

Builder Pattern

Construct complex objects step by step.

```
TestUser user = new TestUser.Builder()
    .withUsername("johndoe")
    .withEmail("john@example.com")
    .withRole("admin")
    .build();
```

6. Static Code Analysis

Use tools like SonarLint to catch issues early (unused variables, potential bugs).

7. Data-Driven Testing

Run same test with multiple data sets.

```
@DataProvider(name = "loginData")
public Object[][] loginDataProvider() {
    return new Object[][] {
        {"user1", "password1"},
        {"user2", "password2"}
    };
}

@Test(dataProvider = "loginData")
public void testLogin(String username, String password) {
    LoginPage loginPage = new LoginPage(initBrowser());
    loginPage.login(username, password);
}
```

8. Exception Handling & Logging

Properly handle exceptions and log information for debugging.

```
protected static final Logger logger = Logger.getLogger(TestBase.class.getName());

public void performAction() {
    try {
```

```
        // Your test action
    } catch (Exception e) {
        logger.severe("Action failed: " + e.getMessage());
        throw e;
    }
}
```

9. Identify Tests to Automate

Focus on repetitive, critical tests. Each test should be independent.

10. Wait Utilities

Use explicit waits instead of hard-coded Thread.sleep().

```
public void waitForElement(WebDriver driver, By locator, int timeoutInSeconds) {
    WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(timeoutInSeconds)
    wait.until(ExpectedConditions.visibilityOfElementLocated(locator));
}
```

11. POJO Classes for API

Use POJOs for type-safe API response handling.

```
public class User {
    private String id;
    private String username;
    private String email;
    // getters and setters
}

String jsonResponse = "{ \"id\": \"1\", \"username\": \"johndoe\" }";
User user = new Gson().fromJson(jsonResponse, User.class);
```

12. DRY Principle

Don't Repeat Yourself. Centralize common code.

```
public abstract class BaseTest {
    protected WebDriver driver;

    @BeforeMethod
    public void setup() {
        driver = new ChromeDriver();
    }

    @AfterMethod
    public void teardown() {
```

```
        driver.quit();
    }
}
```

13. Independent Test Design

Each test should set up and clean up its own data.

```
public class IndependentTest extends BaseTest {
    @Test
    public void testFeatureA() {
        // Test steps independent of other tests
    }

    @Test
    public void testFeatureB() {
        // Another independent test
    }
}
```

14. Avoid Hardcoding

Store configuration in external files.

```
public class ConfigReader {
    private Properties properties;

    public ConfigReader(String filePath) throws IOException {
        properties = new Properties();
        properties.load(new FileInputStream(filePath));
    }

    public String getProperty(String key) {
        return properties.getProperty(key);
    }
}
```

15. SOLID Principles

Follow SOLID for better maintainability. Single responsibility per class.

16. Regular Review & Refactoring

Schedule code reviews. Update libraries. Remove dead code.

17. Reporting

Customize reports with screenshots, logs, and environment details.

```
ExtentReports extent = new ExtentReports();
ExtentTest test = extent.createTest("Login Test");

test.log(Status.INFO, "Starting the login test");
test.log(Status.PASS, "Login successful");

extent.flush();
```

18. Cucumber Reality Check

If not doing full BDD, avoid Cucumber. It adds unnecessary complexity.

19. Choose Right Tools

Select tools based on project needs, not trends. Consider maintenance cost.

20. Atomic Tests

One test covers one feature. Keep tests short and focused.

21. Test Automation Pyramid

Many unit tests (fast) → Some API tests → Few UI tests (slow).

22. Clean Test Data

Create and cleanup data in each test.

```
@BeforeMethod
public void setupTestData() {
    // Code to create test data
}

@AfterMethod
public void cleanupTestData() {
    // Code to delete test data
}
```

23. Data-Driven API Tests

Use realistic data and dynamic assertions.

```
@DataProvider(name = "apiTestData")
public Object[][] apiTestData() {
    return new Object[][] {
        { "param1Value", "expectedResult1" },
```

```
        { "param2Value", "expectedResult2" }
    };
}

@Test(dataProvider = "apiTestData")
public void testApi(String inputParam, String expectedResult) {
    ApiResponse response = apiClient.callApi(inputParam);
    Assert.assertEquals(response.getResult(), expectedResult);
}
```