Automation with Playwright

Day 2

COURSE CONTENTS – DAY 2

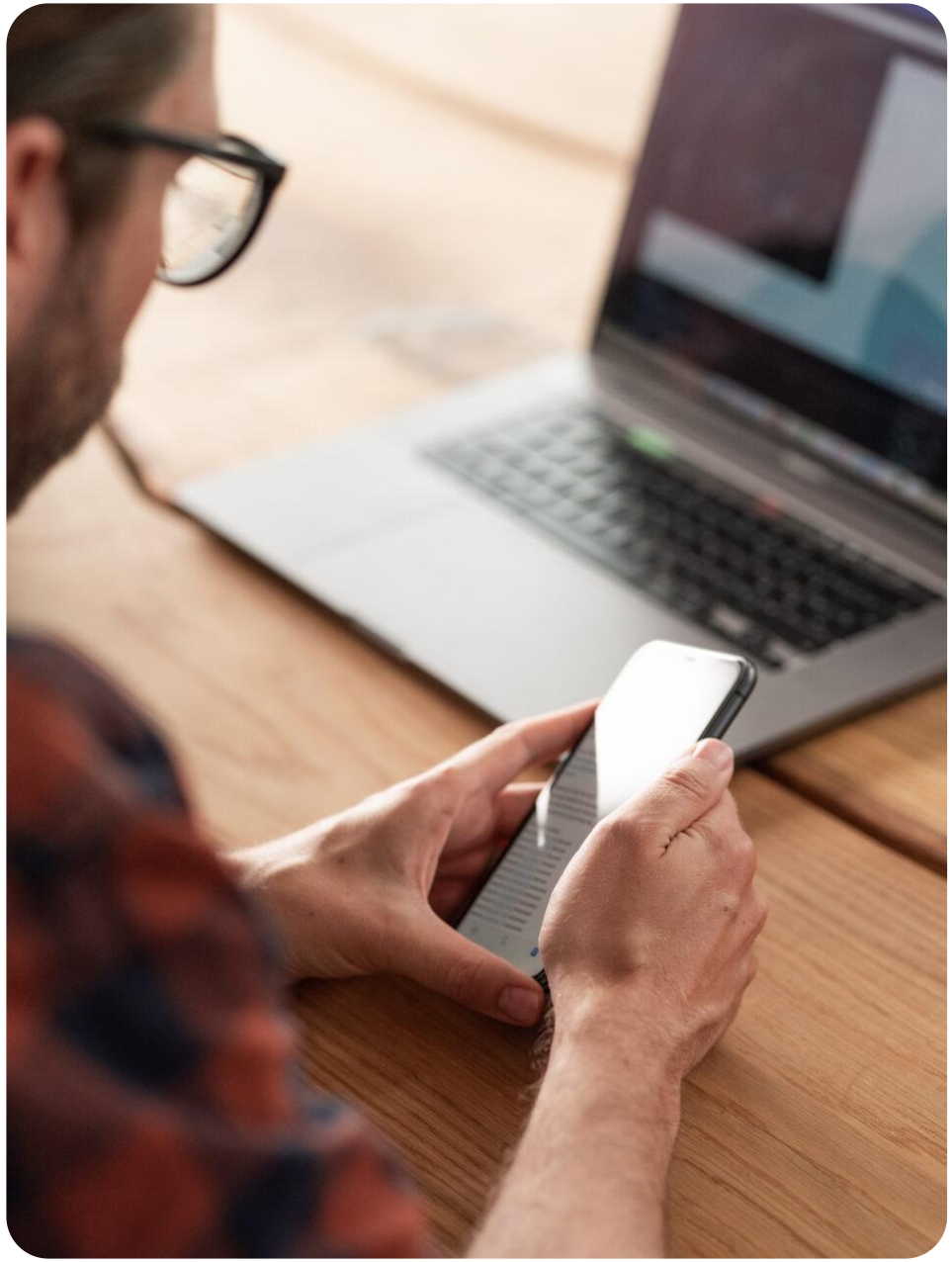
04 Automation in General

05 Automation of Manual Tests

06 Structuring

07 Maintainability

08 Test-first Methodologies



AUTOMATION IN GENERAL

Section Objectives

- Understand the reasons for automation
- Learn what to consider before automation

WHY AUTOMATION?

- Depends on our **Test Automation Goals**.

- Automation may help us:
  - Save time on regression test execution
  - Run tests more frequently (nightly, per PR, on staging, etc)
  - Trigger tests automatically on development updates
  - Reduce feedback time for developers
  - Free testers for exploratory testing and analysis
  - Support overall product quality at scale

Short version
Automation enables faster, more reliable feedback and improves how teams use their testing time

ADVANTAGES OF AUTOMATION

- **Speed and Frequency**
  - Tests run faster
  - We can run more tests
  - We can run tests more often
  - Developers get faster feedback
- **Consistency and Reliability**
  - Results are repeatable
  - No human mistakes
  - Failures and defects are easier to reproduce
  - Stability issues surface earlier
- **Capability**
  - Some tests are too complex to do manually
  - Some tests cannot be performed manually
    - Load testing
    - Performance
    - Parallel workflows
- **Better use of people**
  - Testers can spend time on exploration and risk-based testing
  - Automation supports teams rather than replacing them

DISADVANTAGES OF AUTOMATION

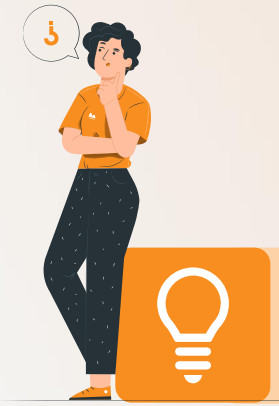
- **Costs**
  - Higher upfront investment
  - Ongoing maintenance costs
  - Test environments, tools and infrastructure needs support
- **Requirements**
  - Requires automation skills and technical expertise
  - Teams must manage tools, frameworks and environments
- **Complexity**
  - Automated tests can become complex systems
  - Processes become more technical
  - Automation introduces its own errors *(false positives, flaky tests, environment issues)*

Key message
Automation is powerful but not free! It requires investiment, skills, and long-term commitment

PLAYWRIGHT IS JUST A TOOL

- **Playwright is one tool in a bigger solution**
  - We aim for a test automation solution, not just a framework
  - Real-world solutions combine multiple tools and technologies
  - Playwright is simply one tool in the toolbox

  - Ask yourself the following:
    - *How does Playwright interact with our existing or future tools?*
    - *Where does it fit in our automation strategy?*



Short version
Don't focus on the tool.
Focus on the solution.

CHOOSING A NEW TOOL

Considerations when selecting a new automation tool

- **Needs and requirements**
  - What problems are we solving?
  - Do we need a single tool or multiple?
  - Requirements:
    - Technical (data, APIs, protocols, integration)
    - Usability and Maintainability
    - Reporting and analysis need
- **Experience and Adoption**
  - Do we have experience with this tool?
  - If not, how quickly can we get it?
  - What is the learning curve?
- **Implementation**
  - Who will implement automation?
  - Do they have the right skills and training?
  - Do they have access to:
    - The right people (developers, DevOps, test analyst)
    - The right infrastructure (system, devices, networks)
    - Necessary accounts and roles

PLANNING AN AUTOMATION PROJECT (1)

- **Objectives**
  - What do we want to achieve?
  - What success looks like (KPI, coverage, ROI, stability)
- **Requirements**
  - Functional and technical requirements
  - Supported technologies
  - Data and environments dependencies
- **Duration**
  - Automation only provides value long-term. (Scripts need time to mature)
- **Interaction with the SUT**
  - Which layers will we automate?
    - Component / API / UI / service
  - Technical interaction
    - Data / Connections / Authentication / Protocols
- **Collaboration**
  - How will developers support automation? (Test hooks, debugging, fixture data, logs)
  - How will testers and developers share results?

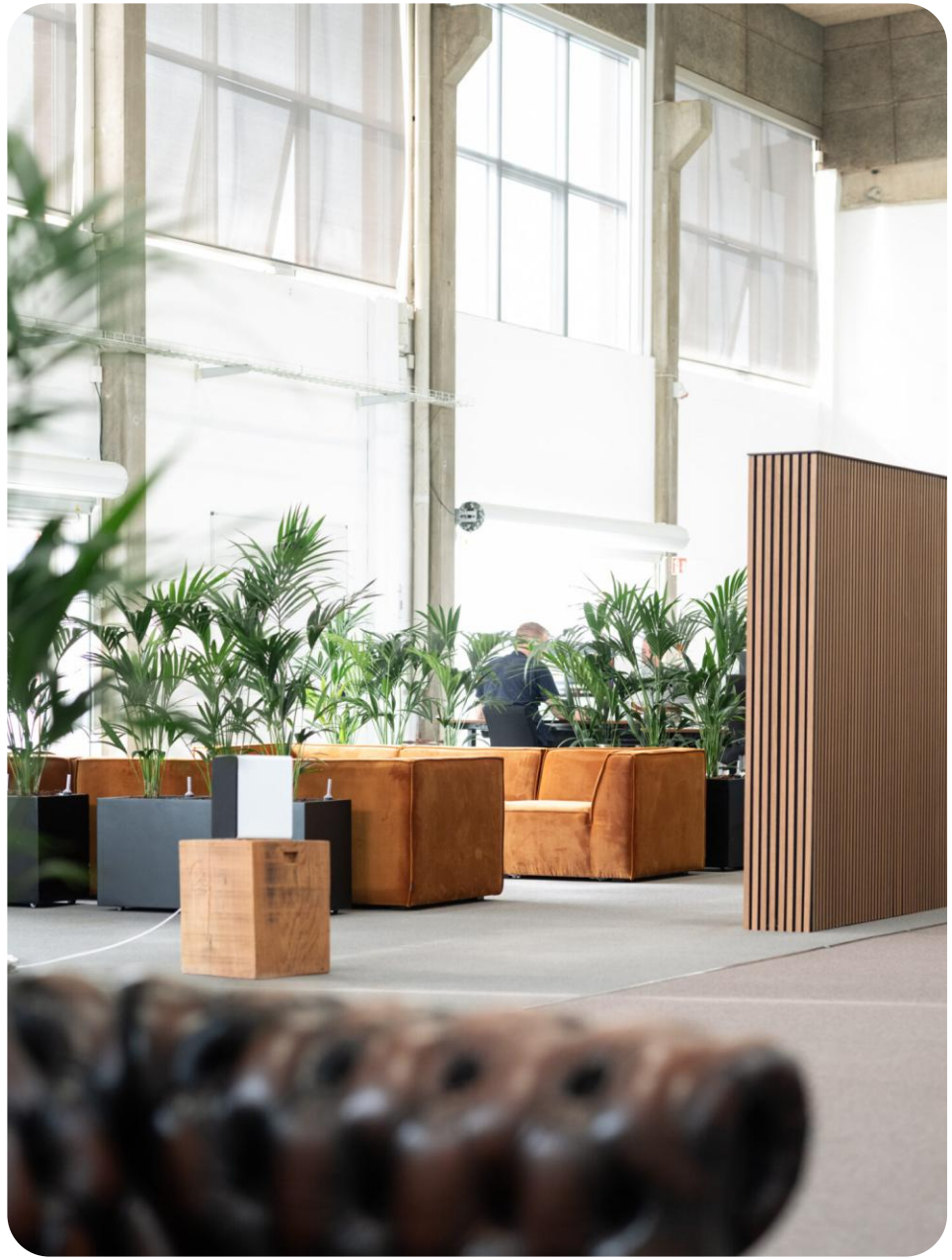
PLANNING AN AUTOMATION PROJECT (2)

- **Test Management**
  - How will we manage the test suite?
  - Who runs the tests?
  - Who monitors the results?
- **Ownership**
  - Who maintains the tests?
  - Who updates them when the system changes?
- **Reporting**
  - How do we report quality?
  - What dashboards or metrics do we need?

AUTOMATION OF MANUAL TESTS

Section Objectives

- Understand how to translate manual test cases into automated tests
- Learn how to use Playwright's Codegen tool to assist in automation



WHICH MANUAL TEST CASES?

Choosing what to automate depends on several factors

- **Frequency**
  - How often is this test executed?
  - Does it run on every release, sprint or deploy?
- **Complexity**
  - How complex is it to run manually?
  - How complex is it to automate?
- **Importance**
  - How critical is this test to product quality?
  - How much value does it provide?

Ideal Candidate
A high-value, frequently executed and manually time-consuming test that is straightforward to automate

REWRITING TEST CASES FOR AUTOMATION

A manual test case is *not* automatically ready to automate

- **What may need to change?**
  - Adjust steps (Manual steps often assume human intuition)
  - Rewrite steps
    - *For precision and repeatability*
  - Resize test cases
    - *Remove overlap*
    - *Make tests reusable*
    - *Support parallel execution*

- **Test Data**
  - Must be explicit, controlled or generated
  - Automation cannot rely on assumptions or "tester intuition"

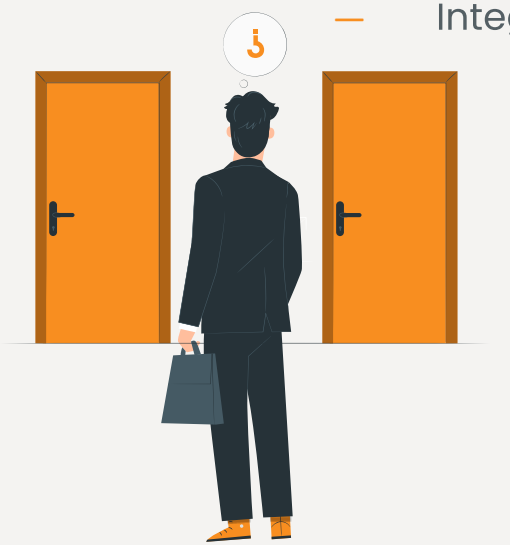
HOW TO IMPLEMENT AUTOMATION

Approach 1 – Capture & Adjust

- Use *capture-replay* tool (e.g., Playwright Codegen)
- Adjust and improve the generated script:
  - Refactor code
  - Verify and optimize locators
  - Handle test data and state

Approach 2 – Develop from Scratch

- Design the automation architecture and strategy
- Implement maintainable tests, page objects, utilities and helpers
- Integrate with CI, environments and reporting



CODEGEN – PLAYWRIGHTS CAPTURE-REPLAY

- **Codegen helps you:**
  - Record user actions
  - Generate a Playwright script as a starting point
  - Experiment with locators and interactions

- **How to start Codegen**
  - From the terminal

cmd

```
playwright codegen URL
```

  - Example

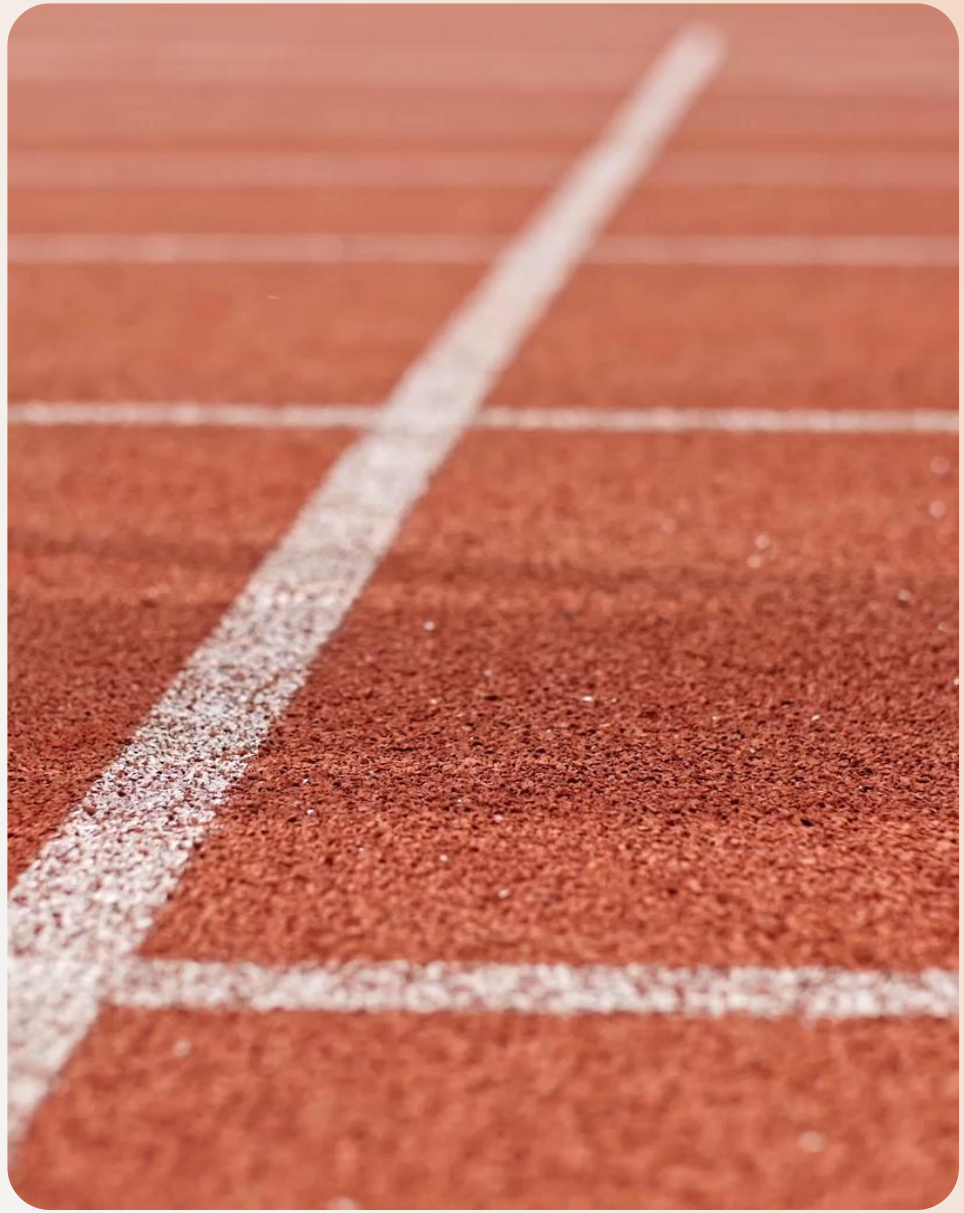
cmd

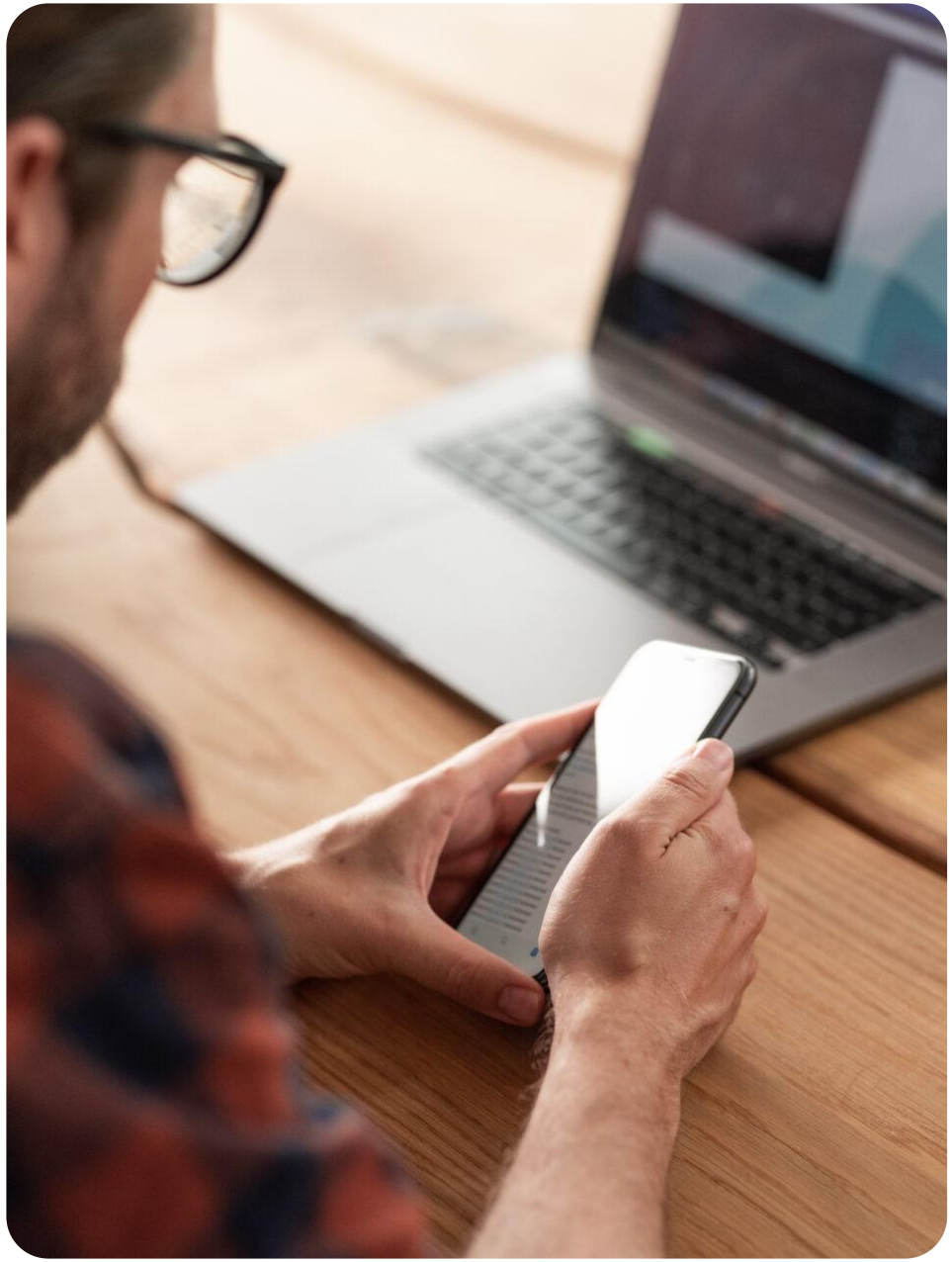
```
playwright codegen https://et.testhuset.dk/
```

EXERCISE 08 (30 min.)

- **Goals:**
  - Evaluate the test cases provided, and determine which is the best candidate for automation
  - Use the Codegen tool to automate the chosen testcase.

- **Locate** 'INSTRUCTIONS.MD'





STRUCTURING

- Understand the strengths and limitations of different test-script design approaches
- Learn how to structure automation using the Page Object Model and other modelling techniques

APPROACHES TO TEST SCRIPTS

— *To build maintainable test automation, we need to choose an appropriate scripting approach.*

— Common approaches include:

- Capture / Replay
- Linear Scripts
- Structured scripts, such as:
  - Keyword-Driven scripts
  - Data-Driven scripts
- Behavior-Driven Development (BDD)

REC

CAPTURE / REPLAY

- A tool records a manual test execution
- The tool can replay those action automatically
- Verification steps may be generated automatically or added manually
- Capabilities vary between tools

Best Practice
If a change in the system under test (SUT) breaks the recording, it's usually easier to record it again rather than try to fix it.

CAPTURE / REPLAY (PROS & CONS)

— **Pros**

- Very low initial effort
- Can generate many scripts quickly
- No programming skills required

— **Cons**

- High maintenance cost
- Limited to no reusability
- Very limited flexibility
- Difficult to control recorded values such as locators, liming, synchronization etc.

LINEAR SCRIPTS

- Tests are recorded manually, then converted into simple scripts
- The scripts can be modified as needed
- The tool can execute ("play") the script
- Many tools allow exporting the script to different programming languages
- Often used when an automation project begins with existing manual test cases
- In practice, linear scripts are usually heavily refactored -> leading to **Structured Scripts**

LINEAR SCRIPT (PROS & CONS)

Pros

- Same advantages as Capture/Replay if kept simple
- Easier to maintain because scripts can be edited
- Useful starting point when existing manual test cases already exists

Cons

- Still poor maintainability as the suite grows
- No true reusability
- Limited flexibility
- Requires the SUT to be complete and stable

STRUCTURED SCRIPTS

- Linear scripts are refactored, or the project begins with a structured design
- Shared functionality is moved into a test utility or helper library
  - Test cases call functions from this library
- Requires some level of coding skills (basic -> advanced depending on structure)
- Primary goal: Improve maintainability and reuse

STRUCTURED SCRIPTS (PROS & CONS)

Pros

- Much higher reusability than previous approach
- Better maintainability
- Greater flexibility
- Faster test creation over time due to reusability

Cons

- Requires programming skills
- Higher initial setup effort
- Still no built-in "business logic" layer
  - Structure may not automatically adapt to changes in requirements

DATA-DRIVEN SCRIPTS

- Test data is stored separately (e.g., files, JSON, CSV, spreadsheets)
- A single "control" test script runs multiple times with different input data
- Useful when test logic stays the same, but input values vary
  - Great for equivalence partitions or large input matrices
- Adding a new test can be as simple as adding a row to the data file

Optional:
Include expected results in the data file, otherwise, expected results must be identical for all test iterations.

DATA-DRIVEN SCRIPTS – EXAMPLE (BEFORE)

Test Case 1

- Open Browser (Firefox)
- Navigate to URL (www.bookme.com)
- Go to Login
- Type Username "JackRyan@domain.com"
- Type Password "PW1234!"
- Click Login
- Click on Make Appointment
- Choose Time Slot 10-11
- Click Book
- Click Logout
- Close Browser

Test Case 2

- Open Browser (Chrome)
- Navigate to URL (www.bookme.com)
- Go to Login
- Type Username "PaulaJones@domain.com"
- Type Password "qwert1#"
- Click Login
- Click on Make Appointment
- Choose Time Slot 13-14
- Click Book
- Click Logout
- Close Browser

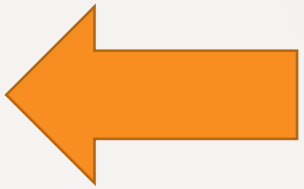
Test Case 3

- Open Browser (Firefox)
- Navigate to URL (www.bookme.com)
- Go to Login
- Type Username "PeterBArnes@domain.com"
- Type Password "zaqwer12"
- Click Login
- Click on Make Appointment
- Choose Time Slot 10-11
- Click Book
- Click Logout
- Close Browser

DATA-DRIVEN SCRIPTS – EXAMPLE (AFTER)

DATA-DREVET TC

- Open Browser (Browsertype)
- Navigate to URL (URL)
- Go to Login
- Type Username (Username)
- Type Password (Password)
- Click Login
- Click on Make Appointment
- Choose Time Slot (From,To)
- Click Book
- Click Logout
- Close Browser



Browser	URL	Username	Password	From	To
Firefox	/home	Testuser1	123455678	10	11
Opera	/home	Testadmin	Admin	11	12
Chrome	/users	Testuser2	12345678	12	13
Edge	/account	Testuser3	87654321	13	14
Safari	/account	Testadmin	Admin	14	15
Firefox	/products	Testuser1	12345678	10	11
Opera	/products	Testuser2	12345678	11	12

DATA-DRIVEN SCRIPTS (PROS & CONS)

Pros

- Same benefits as structured scripts
- Very easy to add new test cases once the control scripts exists
- Non-technical testers can add test data
- Provides broad coverage across input variations

Cons

- Slightly more complex to implement initially
- Control script can grow complex if data requires branching logic
- Risk of missing negative/edge validations if expected results aren't included
- Only valuable when multiple test cases share the same logic
- Data files can grow large or difficult to maintain

KEYWORD-DRIVEN SCRIPTS

- A specialized form of structured scripting
- Common actions are expressed as keywords (high-level commands)
  - A keyword is usually 1-3 words
  - E.g., `login_as_admin`, `open_product_page`, `create_user`
- Keywords represent small pieces of business behavior
- Each test step is written using keywords + parameters
  - Can resemble manual test scripts
  - Keywords are executed by underlaying automation code
- High abstraction level
  - The automation code for each keyword is hidden from the script

KEYWORD-DRIVEN SCRIPTS (PROS & CONS)

Pros

- Same benefits as structured scripts
- Even better maintainability when keywords map to business logic
- Allows non-technical team members to write test steps (once keywords exists)

Cons

- More complex to implement
- Managing large sets of keywords can become challenging
- UI-specific keywords often don't port well between platforms

MODELLING

- For both API and UI tests, it's useful to have a model of the system under test (SUT)
- A model is a simplified, digital representation of a real thing or concept
- In Object-Oriented Programming, classes are often used as models
  - A class typically represents a business entity (e.g., User, Account, Product, Order)
- In a model, business workflows to and from the entity are implemented in system terms

Main Purpose:
Connect business logic with system behavior

PAGE OBJECT MODEL – EXAMPLE (1)

TS pages/LoginPage.ts

```ts
import type { Page } from '@playwright/test';

export class LoginPage {
  constructor(private readonly page: Page) {}

  private usernameInput = this.page.getByTestId('username');
  private passwordInput = this.page.getByTestId('password');
  private loginButton   = this.page.getByRole('button', { name: 'Log in' });

  async goto() {
    await this.page.goto('/login');
  }

  async login(username: string, password: string) {
    await this.usernameInput.fill(username);
    await this.passwordInput.fill(password);
    await this.loginButton.click();
  }
}
```

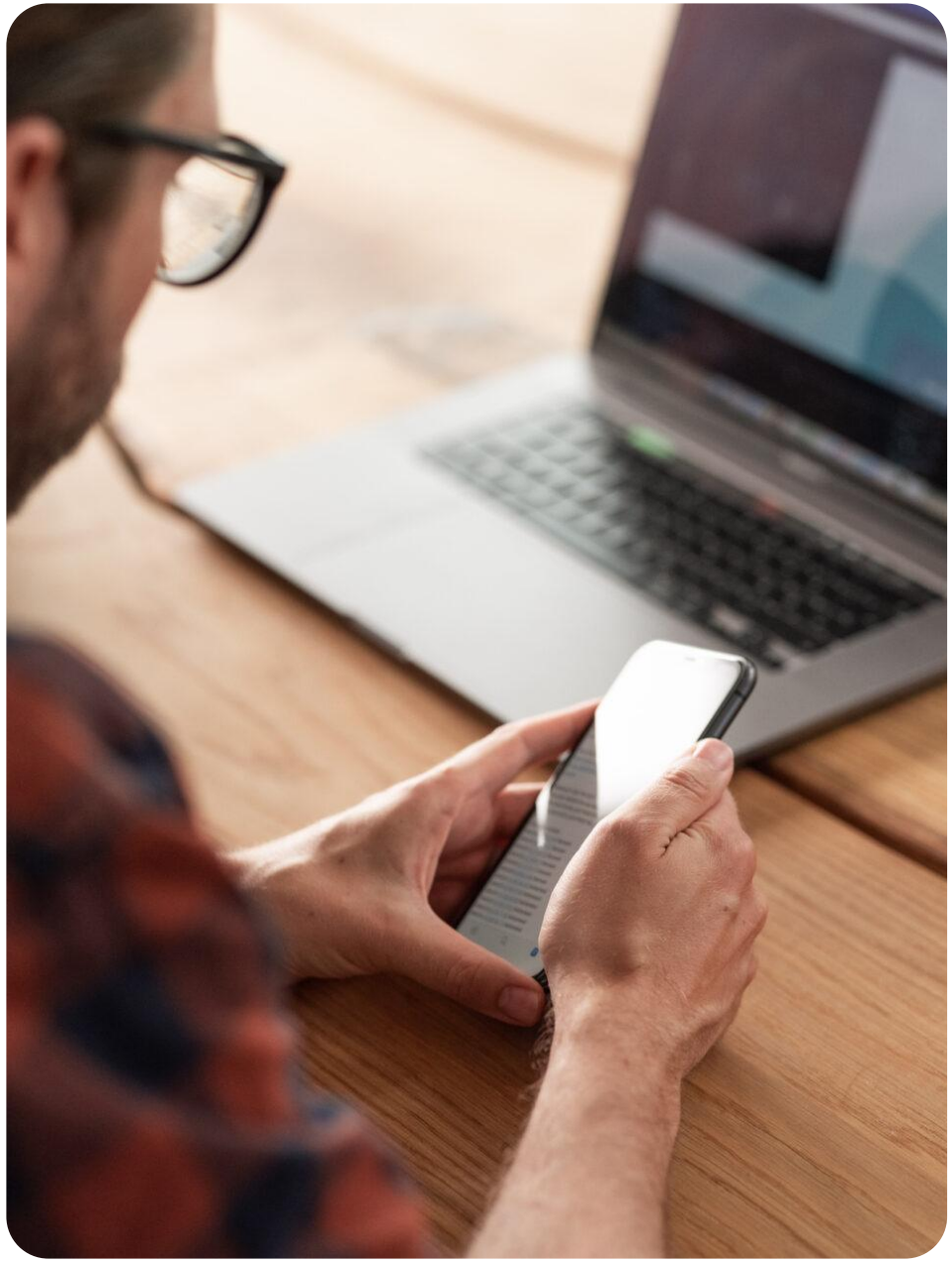
TS landing-page.ts

```ts
import type { Page } from '@playwright/test';

export class LandingPage {
  constructor(private readonly page: Page) {}

  async open() {
    await this.page.goto('/');
  }

  async gotoLogin() {
    await this.page.getByRole('link', { name: 'Login' }).click();
  }
}
```

PAGE OBJECT MODEL – EXAMPLE (2)

TS book-appointment.spec.ts

```ts
import { test } from '@playwright/test';
import { LandingPage } from '../pages/LandingPage';
import { LoginPage } from '../pages/LoginPage';
import { AppointmentPage } from '../pages/AppointmentPage';
import { BookingConfirmationPage } from '../pages/BookingConfirmationPage';

test('user can book an appointment', async ({ page }) => {
  const landingPage = new LandingPage(page);
  const loginPage = new LoginPage(page);
  const appointmentPage = new AppointmentPage(page);
  const confirmationPage = new BookingConfirmationPage(page);

  await landingPage.open();
  await landingPage.gotoLogin();

  await loginPage.login('test_user', 'test_password');

  await appointmentPage.selectFacility('CURA Healthcare Center');
  await appointmentPage.chooseProgram('Medicare');
  await appointmentPage.setDate('2025-12-01');
  await appointmentPage.setComment('Automated test appointment');
  await appointmentPage.bookAppointment();

  await confirmationPage.expectConfirmationVisible();
});
```



MAINTAINABILITY

- Understand why maintainability matters in test automation
- Learn practical ways to improve maintainability in Playwright test suites

MAINTENANCE

Good maintainability supports:
- Reuse of tests and components
- Faster adoption of new SUT features

— Maintenance is a bigger challenge in automation than in manual testing

— When the SUT or its requirements change, we may need to update:
  - Test cases and scenarios
  - Test step implementations (page, objects, helpers, fixtures)
  - Test data and configuration
  - Libraries, language versions, integration and connections
    - These must stay in sync with the SUT and its environment
  - Tools used in the automation pipeline (CI, reporters, runners)
  - Test environments, including mocks, stubs, drivers, emulators, simulators

HOW TO IMPROVE MAINTAINABILITY

- Be mindful of:
  - **Changeability -** how easy it is to adjust tests when the SUT or its requirements change?
  - **Expandability -** how easy it is to add new tests and capabilities to the suite
  - **Adaptability -** how easy it is to run the same tests in different environments (dev, test, prod)
  - **Learnability -** how easy it is for someone new to understand and modify the tests

REUSABILITY

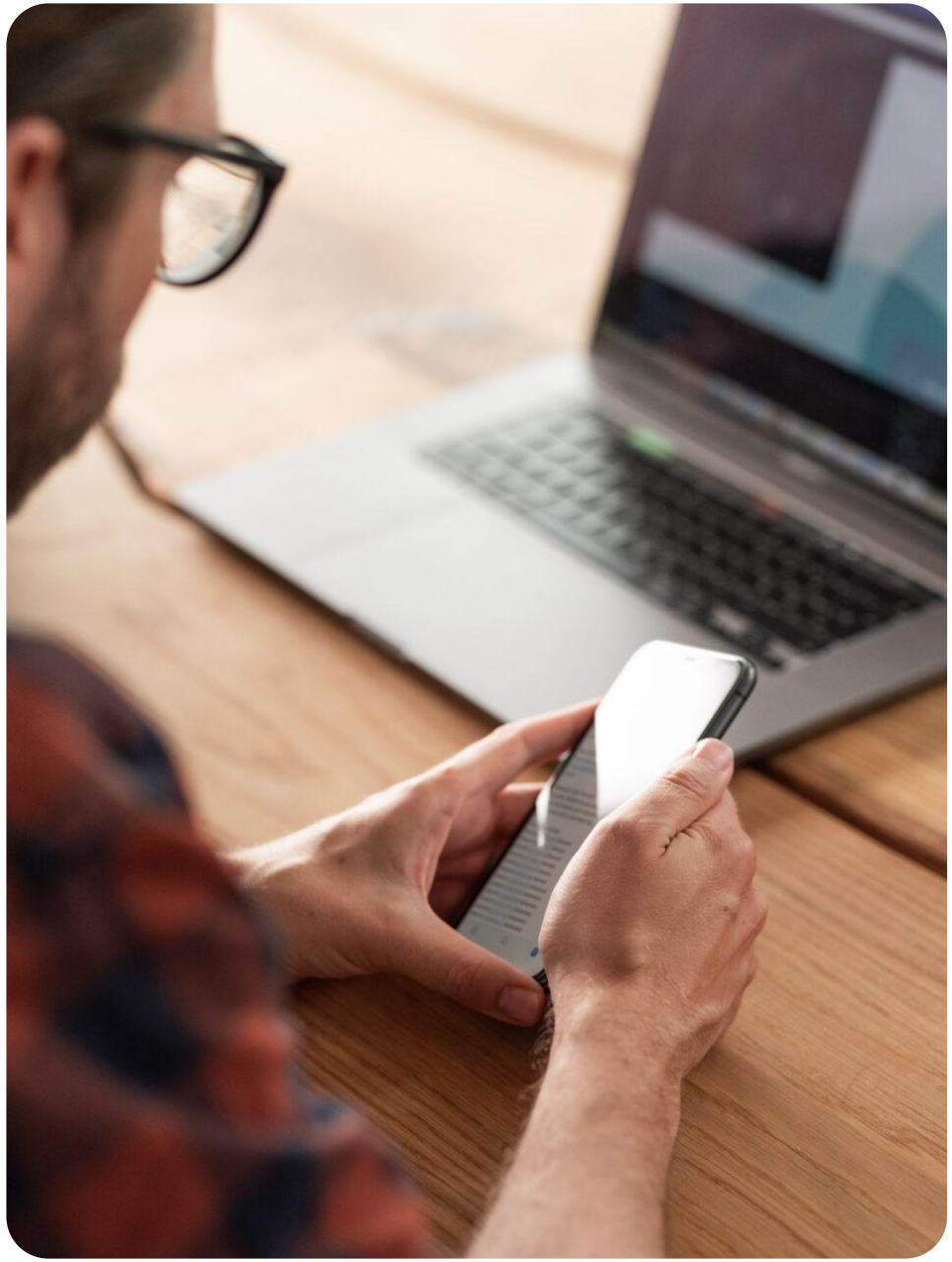
- Our scripting approaches directly affect how much we can reuse
- We should always look for opportunities to increase reuse:
  - If functions or actions are similar, can we generalize them with parameters or shared helpers?
- Good architecture and modelling (e.g., Page Objects, shared fixtures) naturally increase reusability

STANDARDS & CONVENTIONS

- Software is complex; We want to simplify whenever possible
- Agreeing on how we do things gives us:
  - Code that's easier to read and understand
  - Better changes for spotting problems
  - An easier time making changes
- In other words:
  - Better **maintenance**
  - Better **learnability**

Follow clear conventions, such as:

- Naming standards (files, tests, page objects etc)
- Industry or team style guides
- Linting and formatting rules (ESLint, Prettier)



TEST-FIRST METHODOLOGIES

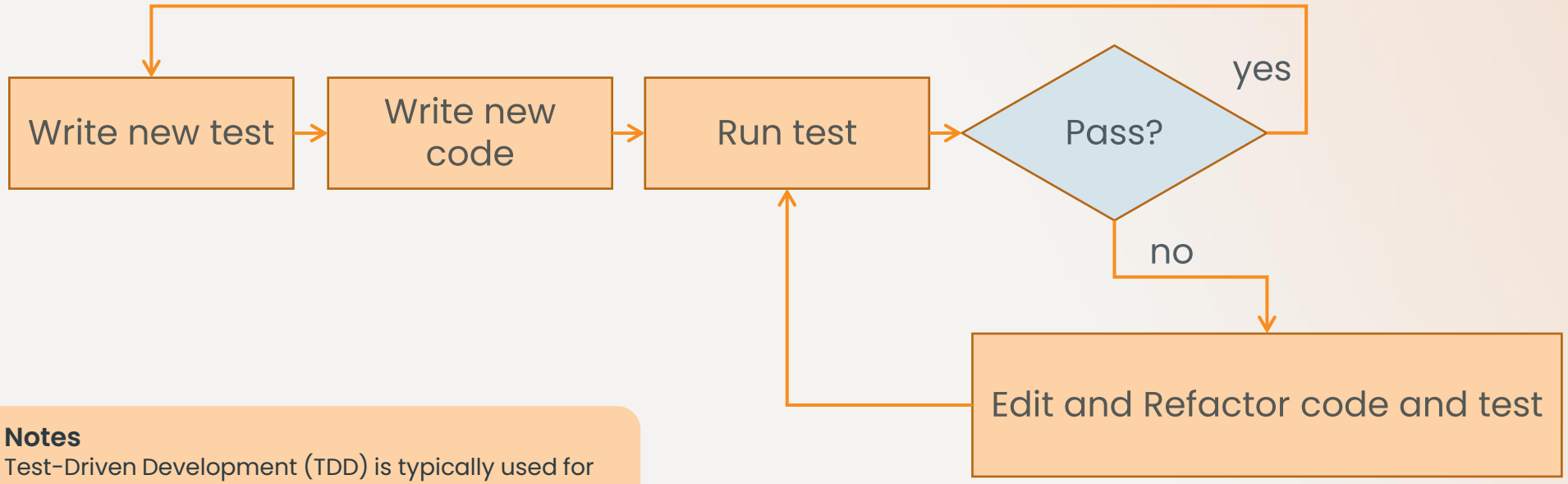
- Understand the purpose of test-first methodologies
- Learn the basics of BDD and implement scenarios using Gherkin

TEST-DRIVEN DEVELOPMENT (TDD)

- TDD is a software development method where tests are written *before* implementation
- Requirements are turned into small, focused test cases that guide coding
- TDD ensures:
  - High test coverage
  - Clear implementation focus
  - A safety net for refactoring and future changes
- Primarily used by developers
- Benefits from good cooperation between devs and testers

TEST-DRIVEN DEVELOPMENT (TDD)

- The TDD workflow:



Notes
Test-Driven Development (TDD) is typically used for unit tests and component-level integration, not UI automation.

ACCEPTANCE TEST-DRIVEN DEVELOPMENT (ATDD)

- ATDD extends TDD by deriving tests from acceptance criteria
- Focuses on business requirements and shared understanding
- Encourages early discussions between
  - Developers
  - Testers
  - Business representatives
- Acceptance tests are created before implementation

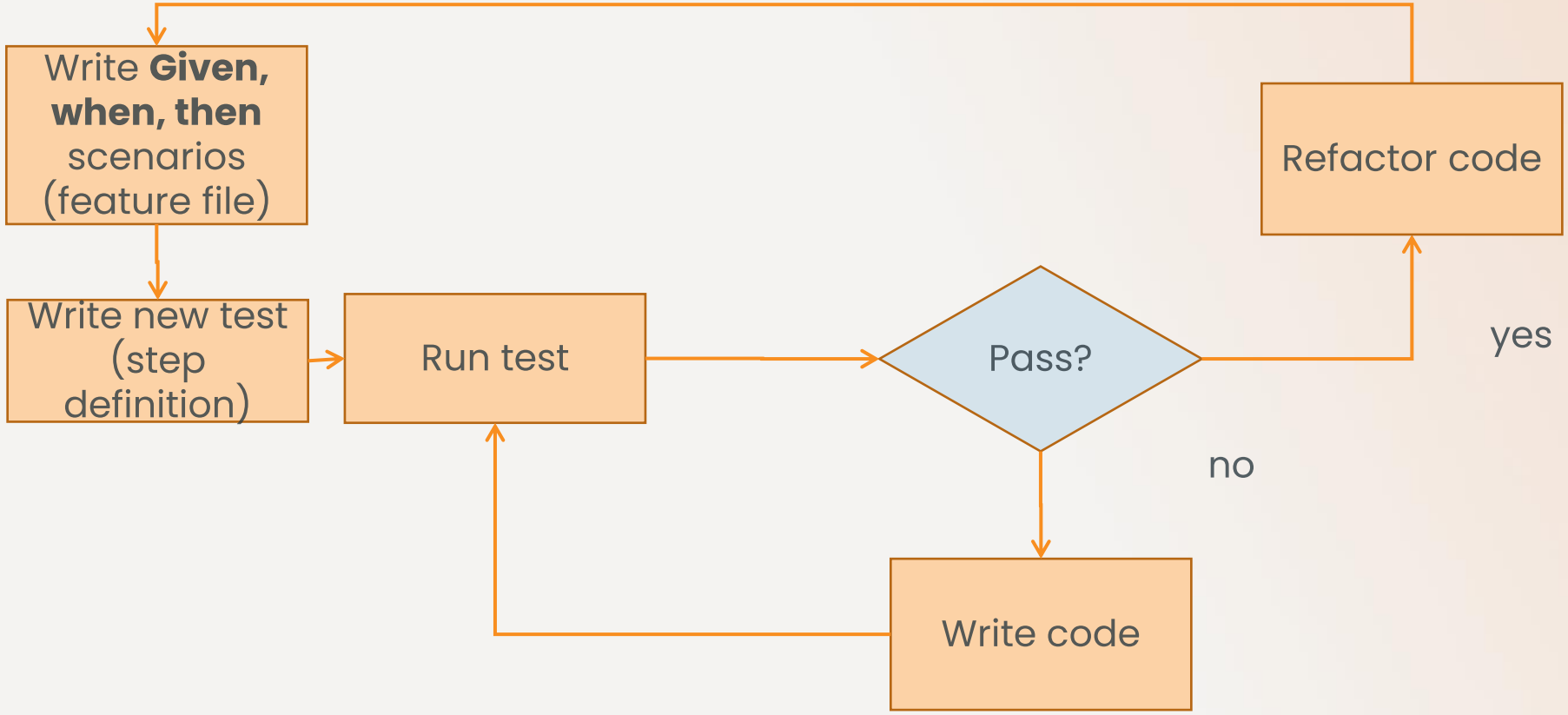
Notes
TDD = tests small units of code
ATDD = tests system behavior from the perspective

BEHAVIOR-DRIVEN DEVELOPMENT (BDD)

- Like ATDD, BDD focuses on business requirements
- Adds the idea of expressing expected behavior through examples (scenarios)
- Scenarios are ideally written collaboratively by:
  - Developers
  - Testers
  - Business
- Scenarios are described using the **Gherkin** syntax
  (e.g., *Given / When / Then*)

BEHAVIOR-DRIVEN DEVELOPMENT (BDD)

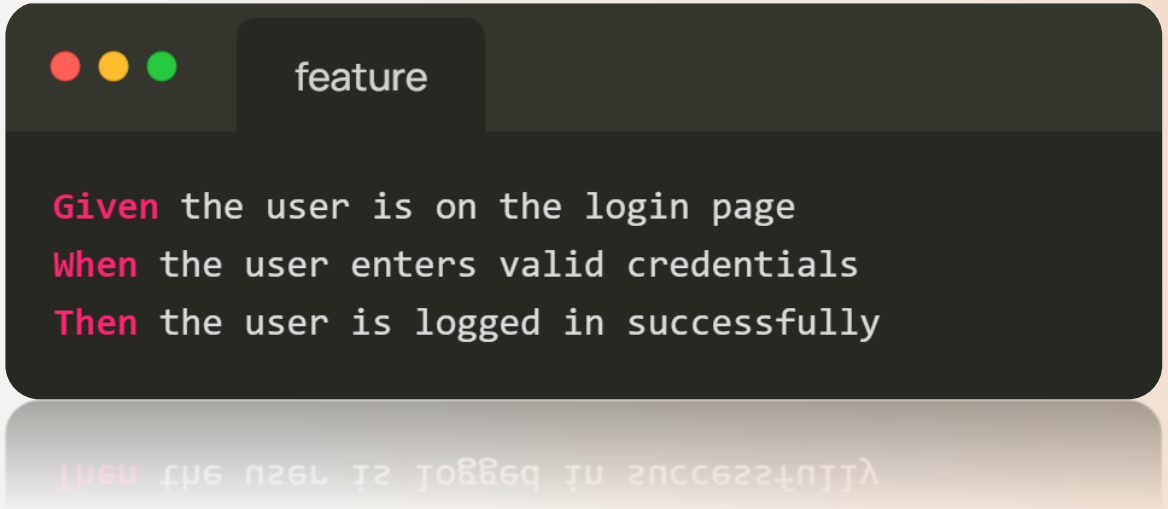
- The BDD workflow:



GHERKIN

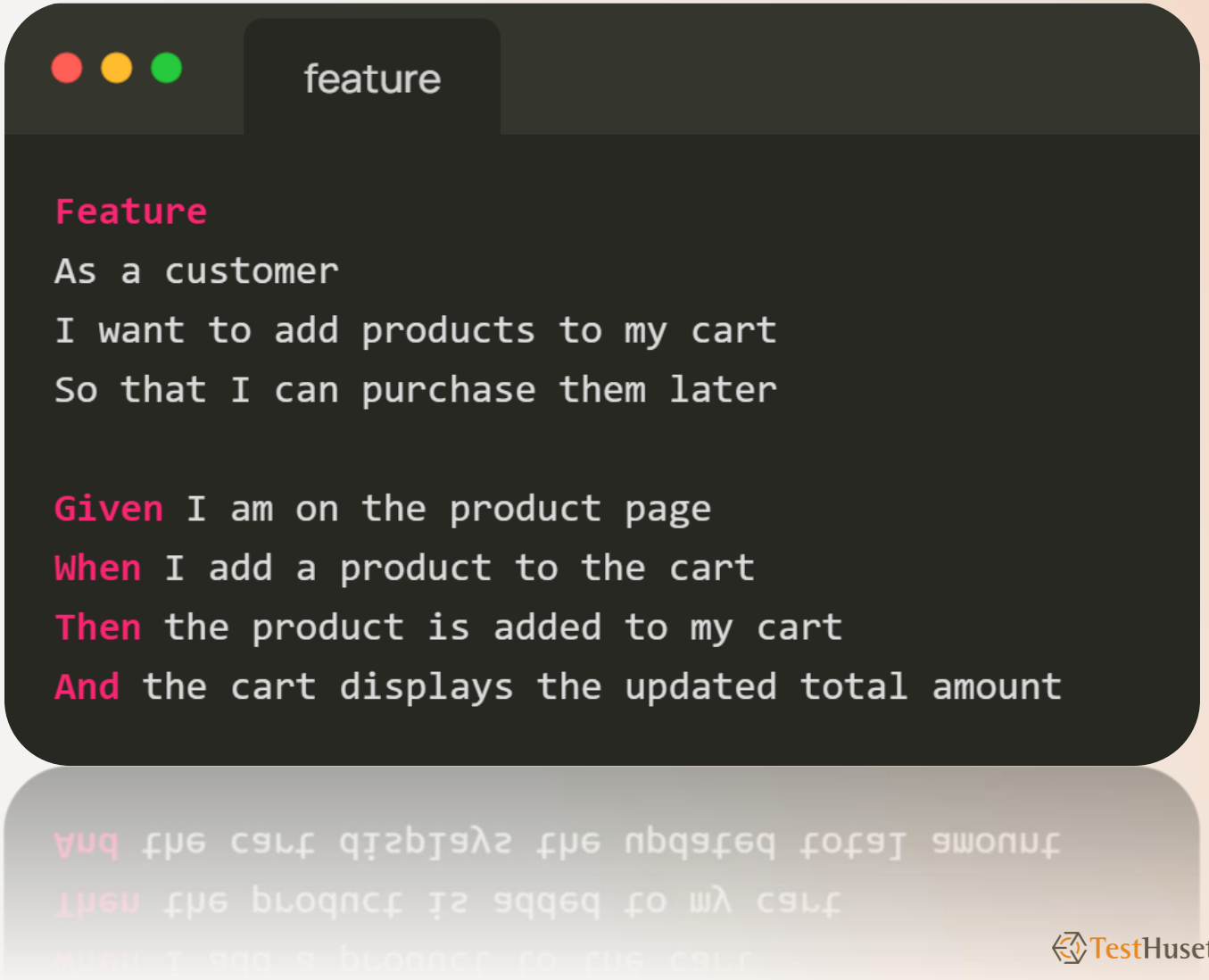
- Gherkin provides a simple, human-readable format for describing test behavior.

- It follows the structure:
  - **GIVEN**: [initial context]
  - **WHEN**: [event that changes the context]
  - **THEN**: [expected outcome]



GHERKIN

— Here's how a user story can be translated into a Gherkin scenario



BDD FRAMEWORK

- Cucumber is not available for Python
  - It's recommended by the Python community to use **Behave** which uses the same conventions
- Feature files live in the `features/` directory
- Step definition files live in `features/steps`
- Environment hooks live in the `features/` directory
  - This is where `before_all`, `before_scenario`, `after_scenario` and `after_all` hooks are defined
- Each scenario receives a shared context object to store data across steps.

EXERCISE 10 (25 min.)

- **Website:** Testhuset
- URL: https://et.testhuset.dk/
  - Choose one of the previous written test cases.
  - Rewrite them as Cucumber (Gherkin) scenario, including:
    - Clear Given/When/Then structure
    - Appropriate step definitions
    - An implementation for each step

- **Locate** 'INSTRUCTIONS.MD'