Fundamentals of BDD with Cucumber (BDD-Cuke)

Syllabus

Version 1.0

Copyright Notice

Thank you to the main author

- Pascal Moll

Thank you to the advisors

- Boris Wrubel and Rahul Verma

Thank you to the review committee

- Kyle Alexander Siemens, Emilie Potin-Suau

Revision History

Version	Date	Remarks
0.01	November 2024	Initial version
0.1	July 2025	Version for Alpha review
0.2	November 2025	Version for Beta review
0.3	January 2025	Adjustments based on reviews
1.0	February 2026	First public realease

Table of Content

Business Outcomes

Business objects (BOs) are a brief statement of what you are expected to have learned after the training.

BO-1	Learn about Behaviour Driven Development in general and what differs from other methods
BO-2	Understand the principles of Agile and Scrum in relation to testing and BDD
BO-3	Understand what the goals of BDD are, and how to achieve them
BO-4	Learn about abstraction and how to use BDD scenarios to your advantage
BO-5	Learn and understand the different testing techniques in BDD and its cycle
BO-6	Unterstand how to setup your testing environment
BO-7	Learn how to use TestNG
BO-8	Get to know the different Cucumber and Gherkin features
BO-9	Learn how to integrate BDD in your Software-Development-Lifecycle
BO-10	Understand how API and Mocks work
BO-11	See the differences between different transmission technologies
BO-12	Learn about karate and how to use it with the gherkin syntax
BO-13	Use and understand Datadriven Testing
BO-14	Learn about the different Mocking functions
BO-15	Get to know about the basics of Continous Integration and Continous Development (DI/CD)
BO-16	Learn about the buildserver Jenkins
BO-17	Generate Testreports and autobuild your tests
BO-18	Bestpractices in all areas

Learning Objectives/Cognitive Levels of Knowledge

Learning objectives (LOs) are brief statements that describe what you are expected to know after studying each chapter. The LOs are defined based on Bloom's modified taxonomy as follows:

Definitions	**K1 Remembering**	**K2 Understanding**	**K3 Applying**
Bloom's definition	Exhibit memory of previously learned material by recalling facts, terms, basic concepts, and answers.	Demonstrate understanding of facts and ideas by organizing, comparing, translating, interpreting, giving descriptions, and stating main ideas.	Solve problems to new situations by applying acquired knowledge, facts, techniques and rules in a different way.
Verbs (examples)	Remember Recall Choose Define Find Match Relate Select	Summarize Generalize Classify Compare Contrast Demonstrate Interpret Rephrase	Implement Execute Use Apply Plan Select

For more details of Bloom's taxonomy please, refer to **[BT1]** and **[BT2]** in References.

Hands-on Objectives

Hands-on Objectives (HOs) are brief statements that describe what you are expected to perform or execute to understand the practical aspect of learning. The HOs are defined as follows:

- HO-0: Live view of an exercise or recorded video.
- HO-1: Guided exercise. The trainees follow the sequence of steps performed by the trainer.
- HO-2: Exercise with hints. Exercise to be solved by the trainee, utilizing hints provided by the trainer.
- HO-3: Unguided exercises without hints.

Prerequisites

Mandatory

- None

Recommended

- Some Agile or Scrum certificate like PSM or CSM or ASF or at least read the Scrum guide.
- Basic knowledge (ISTQB-CTFL or ISTQB-CFTL-Agile Tester) of testing in general.
- Having at least 1 year working experience in Agile and in testing.
- Read a book about Agile Testing like ´Agile Testing´ **[JL1]** and ´More Agile Testing´ **[JL1].**

Chapter 1 - Fundamentals of BDD

Keywords

BDD, Behaviour Driven Development, Data Driven Testing, Keyword Driven Testing

LO-1.1	K1	Understand what Behavior-Driven Development (BDD) is and how it differs from other testing methods
LO-1.2	K2	Be able to name key milestones in the development of BDD.
LO-1.3	K3	Describe the impact of BDD on software development quality
LO-1.4	K2	Understand basic agile principles and recognize their relationship with BDD. Explain the role of feedback loops in agile methods in the context of BDD
LO-1.5	K2	Explain how BDD is implemented in Scrum
LO-1.6	K2	Describe the specific responsibilities of each stakeholder (Product Owner, Developer, Tester) in the BDD process.
LO-1.7	K2	Describe techniques for abstracting requirements
LO-1.7.1	K1	Understand the process of requirement abstraction
LO-1.7.2	K2	Create examples of applying Gherkin syntax in test scenarios
LO-1.7.3	K1	Define the concept of separating business logic from implementation
LO-1.7.4	K2	Identify examples of reusable steps in test scenarios.
LO-1.7.5	K1	Recognize the role of communication between stakeholders in the BDD process.
LO-1-8	K3	Identify examples of test scenarios and be able to choose the right ones based on BDD principles
LO-1.9	K2	Explain the advantages of DDT in the context of BDD, including increased testing efficiency.
LO-1.10	K2	Explain the differences between KDT and DDT and discuss their respective use cases
LO-1.11	K1	Provide an overview of other common tools and frameworks for BDD (e.g., Cucumber, FitNesse)
LO-1.12	K2	Explain the most common challenges and disadvantages of implementing BDD
LO-1.13	K1	Recognize the main reasons for BDD's popularity in software development

1.1 What is Behavior-Driven Development?

Behavior-Driven Development (BDD) is an agile software development method that focuses on collaboration between development, quality assurance, and business stakeholders. The behavior of a feature is described in scenarios that are understandable to all parties. These descriptions can also be used to derive tests. Descriptions, tests, and other necessary information are stored centrally, forming a shared source of truth.

Core Concepts of BDD

- Behavior Specification

  BDD encourages specifying the behavior of an application in a language understandable to all stakeholders. This often takes the form of "Given-When-Then" scenarios that describe a precondition (Given), an action (When), and the expected outcome (Then).

- Collaboration
  BDD fosters collaboration between different stakeholders, including developers, testers, and business experts. This collaboration helps create a shared understanding of requirements, their implementation, and related tests.
- Testing as Part of Development
  In BDD, testing is not treated as a separate step as in other models, but as an integral part of development. Tests are created as early as possible and also serve as documentation of requirements.
- Focus on Stakeholder Needs
  BDD places great emphasis on ensuring that developed features meet the actual needs of users and stakeholders. Scenarios should also be described from the end-user's perspective.
- Iterative Development
  Like other agile methods, BDD works in short iterations to quickly gather feedback and make adjustments early. Scenarios are continuously refined and extended to improve understanding and test coverage.

By applying these concepts, BDD aims to reduce misunderstandings and ensure that the developed software meets stakeholder expectations. In practice, BDD is often used together with Scrum and does not represent an independent collaboration framework, but rather a methodology.

1.2 History of BDD

- Origins of TDD
  Test-Driven Development (TDD) became popular in the late 1990s, especially due to the work of Kent Beck and other members of the Extreme Programming (XP) movement. TDD focuses on writing tests before the actual code is developed, encouraging a test-first mindset and early defect detection.
- 2003
  The term Behavior-Driven Development was coined by Dan North. He observed that software implementation should begin with describing its behavior.
- 2008
  Release of Cucumber, an important BDD tool that allows tests to be written in the easily understandable Gherkin language. Cucumber boosted BDD adoption beyond the Ruby community.
- 2010s
  BDD gained popularity across many programming languages and frameworks. Numerous tools and libraries were developed to support BDD practices (e.g. SpecFlow for .NET).
- 2015 and beyond
  BDD is increasingly seen as part of agile development methods and is applied in DevOps environments as well as microservices development.

1.3 Goals of BDD

- Improved Communication
  BDD encourages collaboration between developers, testers, and non-technical stakeholders (e.g. product managers or customers). Using a shared language (often Gherkin syntax) ensures a common understanding of requirements.
- Early Validation of Requirements
  Writing tests before implementation (test-first approach) helps identify and resolve misunderstandings about requirements early, reducing risks of defects and costly changes later in development.
- Living Documentation
  Specifications in BDD also serve as living documentation of the system. They are easy to understand, accessible to both technical and non-technical team members, and always up to date since they also function as executable tests.
- Testing Behavior
  BDD emphasizes the desired behavior of the software from the user's perspective. This ensures that developed features truly meet user needs and support intuitive usage.
- Enabling Fast Team Response
  BDD supports agile development methods, allowing teams to respond quickly to changing requirements and make iterative improvements.

1.4 Agile Fundamentals with BDD

- Customer Orientation
  BDD encourages close collaboration with stakeholders, making their requirements and expectations easier to understand. This aligns with the agile value of customer collaboration over contract negotiation.
- Iterative Development
  Features are defined and implemented in small increments. Test scenarios are expressed as "Given-When-Then," providing a clear structure for iterative development.
- Transparency
  Using clear, understandable language in BDD scenarios creates transparency for all the people involved. Stakeholders and team members can understand requirements and tests, improving communication and reducing misunderstandings.
- Self-Organizing Teams
  Teams are encouraged to jointly define requirements and tests, fostering ownership and teamwork – a core aspect of agile methods.
- Continuous Improvement
  Regular review of BDD scenarios during retrospectives or sprint reviews helps teams improve

processes and ensure that scenarios continue to meet stakeholder needs. This also involves checking regularly.

- Flexibility and Adaptability
  Since BDD is closely tied to customer requirements, teams can quickly react to changes, add new scenarios, or adjust existing ones. This means that changing situations are always addressed and taken into account.

- Technical Excellence
  BDD promotes good development practices by having all team members write tests. Developers test their code before handover, encouraged to do so in a universally understandable language. The TDD approach is also encouraged, improving code quality. This contributes to technical excellence.

1.5 Applying BDD in Agile Methods

1.5.1 Scrum

1.5.1.1 Integrationg BDD into Scrum

1. Roles

   - Product Owner: Defines requirements in the form of BDD scenarios and ensures that the scenarios are clear to avoid misunderstandings.
   - Scrum Master: Supports the team in implementing BDD practices and promotes a culture of collaboration and continuous learning.
   - Development Team & QA: Work together to turn BDD scenarios into working code, add new scenarios when needed, and involve testers early. Ensure that all perspectives are taken into account.

2. Sprint Planning
   - Plan already created BDD scenarios:

During sprint planning, new features or user stories are planned into the sprint in the form of BDD scenarios. The scenarios can be planned as tasks that still need to be formulated or as scenarios that have already been defined.

- Prioritization:

The scenarios are prioritized and added to the sprint backlog.

3. Sprint Review

Presentation of the implemented features based on the BDD scenarios. Stakeholders provide feedback on the implemented functions and their compliance with expectations.

1.5.1.2 Continuous improvement

- Adjustment of practices:

Based on the findings from the retrospective, the team adapts its approach to increase efficiency and effectiveness in the application of BDD.

- Training and continuous education:

If necessary, training courses or workshops are offered to deepen the team's understanding of BDD and share best practices.

4. Sprint Retrospective

Reflection on the sprint process:

- The team discusses how well BDD practices were implemented during the sprint.
- The team identifies the strengths and weaknesses in dealing with BDD scenarios.
- They discuss opportunities for improvement.
- They discuss which aspects of BDD integration worked well and which did not.
- They develop action points to improve collaboration between developers, testers, and the product owner in future sprints.

5. Daily Stand-up

Team members also report on progress in implementing the BDD scenarios:

- Discussion of challenges or obstacles encountered during implementation.

6. Development Practices

- Test-First Approach: Development begins with writing tests based on the BDD scenarios. These tests serve as specifications for the functionality to be developed.

Collaboration: Developers and testers work closely together to ensure that the implemented features correspond to the defined scenarios.

1.5.1.3 Advantages of Integrating BDD in Scrum

- Improved understanding

Using common language improves communication between technical and non-technical stakeholders.

- Early error detection

Tests are written early, which helps to identify errors during the development phase.

- Higher quality

Close collaboration between developers and testers leads to higher software quality and better feedback.

- Customer focus

Focusing on behavior from the end user's perspective helps ensure that the developed features meet real needs.

1.5.1.4 Disadvantages and Challenges

1. Training period:
    - Education

Teams that are new to BDD need time to become familiar with the concepts and syntax (e.g., Gherkin), which can cause initial delays.

Training may be necessary to ensure that all team members understand and can apply the principles of BDD.

2. Additional effort:

- Test development

Writing BDD scenarios requires additional effort compared to traditional testing methods. This can prolong the development process, especially in the beginning, particularly if the team is inexperienced.

- Maintenance of scenarios

Maintaining and updating BDD scenarios can be time-consuming, especially if requirements change frequently.

3. Misunderstandings during implementation:

- Unclear scenarios

If the BDD scenarios are not clearly formulated or there are misunderstandings among stakeholders, this can lead to incorrect implementations.

- Excessive complexity

Teams may include too many details in the scenarios, making them confusing and difficult to understand.

4. Dependence on collaboration:

- Team composition

The success of BDD depends heavily on collaboration between developers, testers, and subject matter experts. If this collaboration is not harmonious, it can compromise the effectiveness of BDD. A high degree of coordination is always required. Old habits must be changed and replaced with new ones. This is often referred to as the culture of change.

- Customer & stakeholder engagement

Active engagement of customers and stakeholders is crucial to the success of BDD. If stakeholders are not regularly involved, important requirements may be overlooked or implemented incorrectly.

1.5.1.5 BDD & Kanban

- Kanban Principles
  - Visualization: Workflow is visualized to make bottlenecks and progress visible.
  - Work-in-Progress (WIP) Limits: Defined maximums prevent overload and encourage task completion.

1.5.1.6 Integration of BDD into Kanban

- User stories as a starting point

User stories can be represented as tasks in the Kanban board. Each story should contain clear acceptance criteria formulated in BDD style.

- Collaboration

Developers, testers, and business users work together to define the acceptance criteria. This promotes a common understanding of the requirements.

- Test automation

The acceptance criteria can serve as the basis for automated testing, which supports quality assurance throughout the development process.

1.6 Comparison with Scrum

Aspect	Kanban	Scrum
Framework	Flexible, no fixed roles	Strict, with defined roles
Roles	None defined, team members take on different roles	Fixed roles: product owner, scrum master, development team
Planning	Continuous; no sprints	Sprint-based (2–4 weeks)
Work-in-Progress	WIP limits to avoid overload	No specific WIP limits; Focus on sprint goals
Changes	Anytime	Restricted during sprint
Meetings	Ad-hoc	Structured (Planning, Daily, Review, Retro)
Goal Setting	Continuous flow and delivery	Delivery at end of sprint

1.7 Advantages and Disadvantages of Scrum with BDD

1.7.1 Advantages of BDD with Kanban

- Efficient workflow through visualization of the workflow and WIP limits

The Kanban board helps to visualize progress and identify bottlenecks in the workflow.
Limiting work in progress (WIP) ensures that the team focuses on completing tasks before starting new ones.

- Better traceability through transparent requirements and test documentation

The use of BDD makes it easier to make requirements traceable and ensure fulfillment.
The tests themselves serve as living documentation of the requirements and their compliance.

- Flexibility

No fixed iterations or sprints, allowing teams to respond very quickly to changes in requirements. Priorities can be shifted dynamically.

1.7.2 Disadvantages

- Less structure and lack of fixed roles

Kanban does not have defined roles such as Scrum Master or Product Owner. This can lead to uncertainty in terms of responsibility.

- Difficulties with prioritization

The lack of sprint planning can make it difficult to set clear priorities. This makes it more difficult to identify and prioritize the most important tasks for the team.

- Challenges in team dynamics due to a lack of rituals

The lack of regular meetings in Kanban can lead to a lack of communication between team members. Collaboration should therefore be strengthened in other ways.

1.8 Clear Test Scenarios through Abstraction

1.8.1. High-Level Requirements

- High level of abstraction

Requirements are formulated at a high level. The focus is on both behavior and user experience. Technical details are hidden behind the formulated BDD steps.

- Scenarios as abstraction

Scenarios describe general behavior patterns that apply to as many different use cases as possible. This makes steps reusable and hides specific implementation details.

- Contextualization

Requirements are considered in the context of the entire system to ensure that they are relevant in different situations.

1.8.2 The “Given-When-Then” Structure

- Scenarios as general as possible, but as specific as necessary

Scenarios are formulated to represent general behavior patterns rather than describing specific implementation details.

- Reusability

The structure allows similar scenarios to be abstracted and reused. Only the most relevant parts need to be adapted and, if necessary, made universally applicable using variables or data tables.

- Contextualization through Given

The “Given” part can contain several conditions to cover different contexts. This creates an abstract view of different initial situations.

- Flexibility through When

The “When” part can include various actions and logic. This allows different interactions to be incorporated into the test.

- Expectations in Then

The “Then” part defines the expectations for system behavior and the expected result.

1.8.3 Separation of Logic and Implementation

The aim of this separation is to separate the business logic (what the software should do) from the technical implementation (how it is implemented).

This promotes clear communication between developers, testers, and subject matter experts.

1.8.4 Reusable Test Cases

BDD allows scenarios to be designed for reuse — across test cases or projects. This reduces development time, simplifies maintenance, and benefits all dependent scenarios when making updates.

1.8.5 Targeted Communication

BDD fosters clear communication among developers, testers, domain experts, and other stakeholders. Using a shared, plain language keeps everyone involved and aligned, so issues and requirements are easy to understand and discuss. The Gherkin format (Given–When–Then) supports this by expressing requirements in a consistent, readable structure. As a result, misunderstandings are reduced.

1.9 Test Types with BDD

1.9.1 Acceptance tests

Acceptance tests check whether the system meets the defined requirements and acceptance criteria. These tests are often derived directly from the user stories and their acceptance criteria and are a central element of BDD. They help to validate the behavior of the system from the end user's perspective.

1.9.2 Integration tests

Integration tests check whether different modules or components of a system work together correctly. In BDD, the scenarios make it easy to see which modules and components interact with each other. The test report clearly shows whether this check was successful.

1.9.3 System tests

System tests test the system as a whole to ensure that it meets the specified requirements. These tests can be created based on the scenarios formulated in BDD and help to validate the overall behavior of the system. This is where the advantages of reusing existing scenario steps come into play. Existing integration tests can be extended to map entire systems.

1.9.4 End-to-end tests

End-to-end tests simulate real user interactions with the system from start to finish, i.e., from the user interface to the persistence layer. These tests are particularly important in BDD because they cover the entire user experience and ensure that all parts of the system work together seamlessly. User feedback can also be checked as a requirement and mapped in scenarios.

1.9.5. Unit tests

Unit tests check individual components or functions of a system in isolation. Although unit tests are not directly part of BDD, they can still be helpful in ensuring that the smallest building blocks of the code are working correctly. In many cases, unit tests can also be derived from the behavioral specifications and form a scenario within a few steps.

1.9.6. Regression tests

Regression tests ensure that new changes to the code do not affect existing functions or behaviors. Existing BDD scenarios are very well suited for inclusion in a regression test suite when executed regularly. They help to ensure that the system continues to function as expected after changes and illustrate the feedback loop in the process.

1.9.7. Performance tests

Performance tests measure how well a system performs under certain load conditions. While BDD focuses primarily on behavior, system performance is also a crucial aspect. This type of test can also be integrated into BDD scenarios and can therefore be tested as well.

1.9.8. Security tests

Security tests check a system for vulnerabilities and evaluate the threat factor. Even though security is not always the focus of BDD, security aspects can be included in the acceptance criteria. For example, a scenario

could describe how a user can access sensitive data and what permissions are required. Different roles for this test could be transferred in a data table.

1.9.9. Usability tests

Usability tests evaluate the user-friendliness and user experience of a system. Since BDD is highly user-centered, usability aspects can be integrated into the scenarios. These tests help to verify whether the system is intuitive and easy to use.

1.10 Data-Driven Testing (DDT)

Data-driven testing is a testing strategy in which the same test cases are executed with different input values and expected results. It allows functionality to be verified under different conditions without having to write a new test case for each combination. Only by specifying different data sets does the test behavior change.

In BDD, data-driven tests can be implemented by parameterizing "Given-When-Then" scenarios, making scenarios more flexible and reusable (see 2.2.6: Parameters and 2.3.7 Data tables).

The test data is also referred to as data sources and can come from various sources, such as CSV files, databases, or Excel spreadsheets. Separating test logic and test data facilitates maintenance and increases reusability. New test data can be easily added, enabling further test scenarios without code adjustments. Extensions and changes can be made quickly by simply adding or removing data rows.

1.11 Keyword-Driven Testing (KDT)

In keyword-driven testing, test cases are defined by keywords. These represent specific actions or functions. It provides an abstract and user-friendly way to create tests without the need for in-depth programming knowledge.

In BDD, keywords can be used to define the steps in "Given-When-Then" scenarios, improving readability and understanding. These keywords often represent reusable test actions and help keep test logic separate from test data. New keywords can be introduced to cover additional functionality or scenarios, and existing tests can be extended easily by adding or updating keywords. This approach supports both reusability and flexibility.

1.12 Tools and Frameworks

Examples include: Cucumber, FitNesse, Robot Framework, JBehave, Sikuli, JSpec.

1.13 Advantages and Disadvantages of BDD

- Advantages:
    - Common language/shared truth source.
    - Living documentation.
    - Ensures the right product is built.
    - Encourages collaboration and feedback loops.
- Disadvantages:
    - High initial cost and learning curve.
    - Requires strong team harmony and collaboration.

- Demands stakeholder involvement — without active engagement, the initiative is at risk of failing.

1.14 Popularity of BDD

The following factors contribute to BDD's popularity as a methodology. These points were introduced in the previous section and are repeated here only to improve clarity.1.14.1 Readability

The use of natural language in "Given-When-Then" scenarios makes tests understandable for everyone involved, even non-technical people.

1.14.2 Agility

BDD supports agile methods by enabling rapid iterations and easy adaptation to changing requirements.

1.14.3 Reusability

Test scenarios can be easily reused and extended, which facilitates maintenance and promotes clarity.

1.14.4 Collaboration

BDD promotes collaboration among developers, testers, and business departments, leading to better understanding and fewer misunderstandings.

1.14.5 Documentation

The scenarios also serve as documentation of the requirements and behavior of the software. As already mentioned, this documentation is always up to date, and by maintaining a common storage location, a common source of truth is created.

Chapter 2 – Test Automation with Cucumber

References

[MS1] [CC1]

Keywords

Gherkin, Datatables, TestNG, Feature Files, Step Files

LO-2.1	K1	Be able to set up a test environment for Cucumber and TestNG.
LO-2.2	K2	Demonstrate the benefits of TestNG for test management and the execution of test scenarios
LO-2.3.1	K3	Enable participants to create their own feature files and apply them in a practical example
LO-2.3.2	K3	Cucumber: Be able to write their own step definitions and link them to their feature files
LO-2.3.3	K2	Cucumber: Explain how both types of tests can be implemented in Cucumber and what to consider
LO-2.3.4	K1	Cucumber: Describe the role of hooks in Cucumber and how they are used to define execution order in test scenarios
LO-2.3.5	K2	Cucumber: Discuss various configurations for runner classes, including selecting specific features or tags
LO-2.3.6	K3	Cucumber: Be able to create parameterized test scenarios in Cucumber. Learn how to define parameters in Gherkin scenarios and use them in step definitions to develop flexible and reusable tests
LO-2.3.7	K3	Cucumber: Learn how to implement data tables in their feature files. Practice creating test scenarios with data tables and learn how to use structured data effectively in Cucumber
LO-2.3.8	K3	Cucumber: Promote understanding and application of complex data tables. Create scenarios with nested or multi-dimensional data structures and learn how to use these in Cucumber

2.1 Setting up the Test Environment software

The test environment forms the basis for automated tests. It includes all components needed to develop and execute tests. This environment contains tools for automation and the necessary components to access the System under Test (SuT).

For this syllabus, the test environment consists of Java, TestNG, Maven, and Cucumber.

- Successful installation of Java and an IDE is assumed.
- TestNG: A popular Java test framework that supports unit and integration testing, with plugins for IDEs like IntelliJ or Eclipse.
- Maven: A build management tool that simplifies handling dependencies and libraries through its central configuration file (POM.xml).

- Cucumber: Completes the setup. Gherkin syntax support is provided via plugins (available in IDE marketplaces), offering syntax highlighting and direct execution of feature files. Dependencies are retrieved from the Maven Central Repository.

2.2 TestNG

[TE1]

TestNG is a comprehensive test framework for Java. It offers a variety of annotations and functions to facilitate the testing of applications. Below are some of the most important annotations and concepts in TestNG:

2.2.1 @BeforeClass

Runs once before all tests in a class, often used to start browsers or set up database connections.

2.2.2 @BeforeMethod

Runs before each test method, e.g., resetting variables or creating new instances. This is ideal for tasks such as resetting variables or creating new instances of objects that are needed for each test.

2.2.3 @AfterClass

Runs once after all tests, ideal for cleanup (closing connections, shutting down resources). This method is ideal for cleaning up resources or closing connections.

2.2.4 @AfterMethod

Runs after each individual test method, ensuring clean state between tests. For example, when resetting a browser status.

2.2.5 Test Groups

- TestNG lets you organize tests into groups. You can then choose to run or exclude entire groups in a single configuration. This is especially helpful in large projects with many test cases, because it allows you to execute only the tests that matter for a given purpose. A common example is creating a smoke test group that runs quickly to confirm the build is stable.
- To define a group, use the groups property in the @Test annotation. These groupings can be specified during test execution.

2.2.6 Parameters

- Parameters can be passed via XML configuration and accessed with @Parameters.
- Useful for running tests with different input values.

2.2.7 Assert Methods

- assertEquals: Checks equality of two values.
- assertNotEquals: Ensures two values are different.
- assertTrue / assertFalse: Check boolean conditions.

TestNG provides structure, efficiency and reliability in testing, which improves both software quality and maintainability.

2.3 Cucumber

Cucumber is one of the best-known BDD tools, using Gherkin syntax to describe test scenarios in plain, readable language.

As already mentioned, Cucumber is one of the best-known BDD tools. It uses the Gherkin syntax to describe test scenarios. This syntax was already described in Chapter 1 and is easy to read for both development staff and specialists without a technical background. With Cucumber, these test cases can be executed. The scenarios are stored in feature files.

2.3.1 Feature files

Feature files are simple text files with the extension .feature. They are written in Gherkin syntax and contain both requirements and associated test data. A scenario reflects a test case. They are used to describe the functionalities from the end user's point of view. These steps are linked to so-called steps and thus enable an association with executable program code. With the most common Cucumber plugins for IDEs, step definitions that have not yet been implemented are created during execution.

2.3.2 Step Definitions

Step definitions, also known as "glue code", are executable code that links the steps in the feature files to the actual implementation. Each step in a feature file is represented by a step definition. It describes exactly what should happen when this step is executed. Within each step, there is associated program code with actions and possible verification. This means that frameworks and libraries from other providers can also be easily integrated into the tests.

2.3.3 UI & Functional Testing

UI testing libraries such as Selenium or Playwright can also be easily integrated into step definitions. The UI tests focus on user interaction with the application, while the functional tests ensure that the business logic is working correctly. The process description for these tests can clearly show the business department exactly which actions are performed and what a workflow looks like.

2.3.4 Hooks

Hooks are special functions in Cucumber that perform actions before or after the execution of test scenarios. They are comparable to the before and after annotations in TestNG and help automate recurring tasks such as loading test data or cleaning up resources. A typical example is starting the browser or opening a frequently used URL before the actual test begins. Unlike TestNG, hooks are executed at the scenario level. Three before hooks and three after hooks are considered essential. The BeforeAll hook is useful for performing a one-time action before scenarios run. The Before hook, in contrast, is executed once before the first step of each scenario, while BeforeStep is executed before every individual step. The same principle applies to the After hooks. In addition, sequences can be defined using order = <sequence number>, which allows multiple hooks to run in a defined hierarchy.

2.3.5 Runner Classes

Runner classes are Java classes that are responsible for executing Cucumber tests. They allow developers to select specific features or tags and define different configurations for the test run. These configurations can be used to integrate plugins, create test reports, or exclude certain test groups via tags. The paths of the feature files and the glue code must be specified within this runner class. These can also be specified using the plugin option.

2.3.6 Parameterization

Parameterization enables test scenarios to be executed with different input values. This eliminates the need to duplicate or adapt the code. This is comparable to the use of variables in a programming language. The "content of a variable" is defined in the feature file and transferred to the glue code. A variable is inserted at the appropriate place in the step file and can be used in the following program code of the method.

In addition, the "Background" keyword can be used to define common steps for all scenarios in the same feature file. This keyword ensures that the defined steps within each scenario are executed before the first

scenario step. This eliminates the need to repeat these steps for subsequent scenarios. This makes the test code clearer and easier to maintain, as all scenarios can refer to a uniform starting point.

2.3.7 Data Tables

Data tables in Cucumber are a way to use data in a structured way in test scenarios. This data can be multiple input values as well as expected results. This information is specified in tabular form in the feature file. Data tables can be placed under a respective test step or at the end of the scenario. Each individual table element can also be accessed in the program code and thus influence the sequence of events.

The added value of data tables lies in the fact that they improve the reusability and maintainability of test scenarios, as a separate scenario is not required for each combination of inputs. Instead, multiple data sets can be combined in a single table, which reduces the testing effort and increases coverage at the same time.

2.3.8 Complex Data Tables, Hybrid Forms

More complex data tables require the “Scenario Outline” extension and the “Examples” keyword.

This tells Cucumber to execute the scenario using the example table provided under Examples. The table headings define the available columns. These headings can then be referenced in the scenario using angle brackets. During execution, the placeholder is replaced with the corresponding value from the table. One scenario run is performed per line. It is also possible to combine data tables with example tables. This makes it possible to use a table with variables within a scenario step. If another test run with different data is needed, it is enough to add a new row to the table.

Chapter 3 – The BDD Cycle

Keywords

TDD (Test-Driven Development), ATDD (Acceptance Test-Driven Development)

LO-3.1	K2	Understand how BDD improves collaboration between developers and business stakeholders.
LO-3.2	K1	Recall the definition and main characteristics of TDD.
LO-3.3	K3	Describe scenarios where each method is best suited.
LO-3.4	K3	Create a simple example for each phase of the BDD cycle and explain how these are applied in a project.
LO-3.5	K2	Understand why specific pros and cons are relevant for different projects.

3.1 Motivation for the Behavior-Driven Development Cycle

BDD aims to strengthen collaboration between developers, QA, and stakeholders. Earlier chapters introduced BDD mainly in the context of creating specifications. However, if stakeholders only contribute to writing specifications and are not involved in testing or development, valuable opportunities for alignment and feedback are lost.

Stakeholders should also contribute to:

- Test case creation
- Result analysis
- Feature development support

They are not expected to program, but their insights can prevent misunderstandings.

Example: A stakeholder reviews a test report and validates test steps against the feature vision. This directly highlights potential gaps.

All participants should be involved in every step of the development process. This ensures that:

- Everyone understands the specifications.
- It becomes easier to verify that the right product is built, and built correctly.
- Collaboration stays continuous throughout the project.

3.2 Recap: Test-Driven Development

TDD is a method where tests are created before the actual code is written.

Process (3 steps):

- Write a Test: Define what the code should do.
- Implement the Code: Write the minimum code needed to pass the test.

Refactor: Improve the code without changing functionality.
This sequence helps to identify errors early on and implement requirements in a targeted manner.

Advantages

- Early defect detection due to test-first approach.
- Improved maintainability (tests always available for verification after changes).
- Prevents forgetting tests, as they are the first step.

Disadvantages

- Requires adjustment and can be time-intensive at the start.
- Overhead for small or trivial projects.
- Risk of false sense of security (tests may not cover all cases).
- Learning curve involved in adopting the method.

3.3 Differences between BDD, TDD, and ATDD

Acceptance Test Driven Development (ATDD for short) is a process in which acceptance criteria and acceptance tests are created before the actual implementation. Developers, testers, and the specialist department work together to formulate the requirements in the form of tests. These specifications are intended to describe the desired behavior of the software. The aim is to ensure that all specified requirements are met.

3.3.1 ATDD Development Process

The process begins with requirements analysis. Here, developers, testers, and the business department work together to define both criteria and test scenarios for a feature. After the analysis, acceptance tests are defined. These tests check the desired behavior of the feature. This is followed by the actual implementation of the feature. Finally, the tests are executed. Once this phase has been successfully completed, optional refactoring can take place, or the functionality is considered implemented. In case of failure, the necessary code corrections are made.

3.3.2 Advantages and Disadvantages of ATDD

Advantages of ATDD

- Better communication.
- Early identification of misunderstandings.
- Requirements serve as a living reference.

Disadvantages of ATDD

- Requires strong collaboration.
- Clear and precise wording takes time.

3.3.3 Comparison: BDD vs. TDD vs. ATDD

Aspect	TDD	BDD	ATDD
Focus	Code functionality	Desired System behavior fullfilled	Stakeholder requirements
Scope	Unit/module level	Entire system	Acceptance tests & user needs

Aspect	TDD	BDD	ATDD
Test Language	Technical, in code	Natural language (often Gherkin)	Natural language
Participants	Developers	Developers, testers, PO	Developers, testers, stakeholders
Collaboration	Low (mainly devs)	High	Very high
Use Cases	Functions, modules, components	Unit, API, UI, end-to-end	Requirement validation

3.4 Behavior-Driven Development Process

The BDD development process builds on the processes described above.

First, the desired behavior is specified using the Given-When-Then syntax. These specifications can also be used to generate the glue code. As in the TDD approach, the first tests are written next, but they will fail until the corresponding production code is implemented. Once the test cases pass ("green"), the optimization and refactoring phase begins. The cycle then either ends or starts again with further improvements.

3.5 Advantages and Disadvantages

This process again places great emphasis on cooperation between the various departments, without which implementation would be difficult. This approach provides very quick feedback, shows whether the right thing has been developed correctly, and supports the living documentation behind BDD. Developers gain a certain degree of certainty that they are implementing a feature correctly and that it has been tested. Refactoring changes are also covered and can be carried out almost without risk. All parties are involved from the beginning to the end of a feature.

As is well known with BDD, there are high initial costs and a corresponding learning curve. The necessary coordination is very high and must be accepted by all sides. Without the cooperation of all the parties involved, it will be very difficult to implement this process.

Chapter 4 – APIs, Mocking & BDD with Karate

Key Terms

APIs, Mocking, JSON, SOAP, REST

LO-4.1	K2	Describe the basic concepts of APIs and mocks and explain the differences between REST and SOAP.
LO-4.2	K3	Name the main features and architecture of Karate and explain the installation and setup process in a Maven project.
LO-4.3	K3	Understand the concept of data-driven testing, can create such a test using different data, and analyze advantages compared to static tests.
LO-4.4	K1	Understand the role of hooks in the testing process.
LO-4.5	K2	Understand the importance of mocking in the context of API testing and can implement mocking functions in Karate.
LO-4.6	K1	Understand common challenges in API testing, best practices, general know-how, and pros and cons.

4.1 API & Mock Fundamentals

4.1.1 What are APIs?

[MS1] [AW1]

API stands for Application Programming Interface. This is a type of interface that enables different software programs to communicate with each other. APIs enable both data and functions to be exchanged among different systems. For example, a weather app can use an API to retrieve current weather data from a server.

4.1.2 How APIs Work

First, a request is sent to a server. This request is processed and then a response is sent back. This transmission often takes place via the HTTP protocol and can be either synchronous or asynchronous.

4.1.3 API Architecture

An API consists of endpoints through which available functions can be accessed. These endpoints are URLs and can be called up. Parameters can also be used in such calls. Data is exchanged using the JSON or XML format. In addition, authentication mechanisms and authorizations are often in place. Data transmission can be carried out using various HTTP methods, which are discussed in section 4.1.4.

4.1.4 Transmission Standards

- REST
- REST communication takes place via the HTTP protocol and supports both JSON and XML formats. These types of interfaces are very flexible and have few specifications. Transmission takes place directly.
- SOAP interfaces are very strictly designed and adhere to standards, which makes them comparatively rigid. Transmission is carried out with validation against a schema file, also known as WSDL. SOAP only uses the XML format for exchange.

4.1.5 HTTP Methods (focus on 4 main ones)

- There are numerous HTTP methods, for this curriculum, the focus will be on the four most important ones: GET, POST, PUT, and DELETE.

4.1.5.1 GET

- GET requests are used to retrieve data from a server. They do not allow data manipulation.

4.1.5.2 POST

- This type of request sends information or commands to a server. This allows actions to be triggered or processing to be controlled.

4.1.5.3 PUT

- PUT requests enable data to be changed, similar to updating a database. Data sets can be updated or replaced.

4.1.5.4 DELETE

- DELETE requests can be used to permanently remove data from the server.

4.1.6 API Validation

Checking API responses is essential to ensure that the returned data complies with expected standards and formats. Below are some methods that can be used to validate API responses:

4.1.6.1 Schema validation

Uses JSON schema or XML schema to define the structure of the response. These responses are compared with existing XSDs (for XML), for example.

4.1.6.2 Type checking

Checks whether the data types of the fields in the API response correspond to the expected types (e.g., string, integer, Boolean).

4.1.6.3 Value range check

Ensures that value ranges have been adhered to. If specified numerical values are present, strings are within predefined minimum and maximum lengths.

4.1.6.4 Presence of required fields

Checks whether all required mandatory fields are present in the response.

4.1.6.5 Content validation

Validates the content of certain fields, e.g., whether an email address is in the correct format or whether a status value contains a specific set of permissible values.

4.1.7 Mocking

Mocks are replicas or placeholders for real APIs. They are used in development to test functionality without having to access the actual API. This is particularly useful if the real API is not yet ready or if real access to this API is associated with high costs. This means that tests can also be carried out in isolation without

affecting other parts of the system. With mocks, a predefined result can be reliably generated, independent of external influences.

4.2 Karate

[KA1]

Karate is a BDD framework that was developed specifically for API testing and web service testing. It uses the Gherkin syntax described above to write tests in a readable form. HTTP requests are supported directly, enabling easy test case development for APIs. Karate offers functions for validating both JSON and XML responses directly, which enables easy data processing. By setting a configuration variable, parallel test execution is also very easy, which also increases the test execution speed. Karate also has integrated reporting functions for analyzing test results. No glue code is necessary for these API tests; they work out of the box using the Gherkin syntax.

Other third-party systems such as Playwright, Spring, or Xray can also be easily integrated.

4.2.1 Karate Architecture

Karate itself is based on the concepts of Cucumber, but uses its own unique architecture. The key components are web browser automation, API test doubles, and API testing. All three offer numerous interfaces and adapters for running tools such as Playwright or Selenium and controlling them via a runtime environment. Although Karate is primarily designed for API testing, a wide variety of tools can be integrated. API test doubles enable mock server integration. These parts are part of the core component. An optional part enables the integration of desktop test automation tools.

4.2.2 Comparison between Karate and Cucumbe

Unlike Cucumber, Karate already has common frameworks such as Selenium integrated and can be used without any further action. The Gherkin syntax is supplemented by additional features. The HTTP request and response features are particularly important here. Nevertheless, the Gherkin syntax is the same. Karate also offers integrated reporting and generates a report with corresponding error descriptions after each test run. This report shows directly at which step an error occurred. No separate glue code is necessary for Karate.

4.2.3 Karate Runner & Configuration

Karate itself can be integrated via Maven. JUnit can be used to execute Karate tests. However, this requires specifying the classpath of the associated feature file, which is done via Runner.Path. This method also allows tests to be run in parallel, thereby shortening the runtime. Additional configuration options are also available within the runner.

4.2.4 Karate & IDE Plugins

Plugins are available for both Visual Studio Code and IntelliJ to enable feature files to be executed directly in the IDE. These plugins have both free and paid features. There is no plugin for Eclipse, but feature files can still be executed using the standard Cucumber plugin. However, this requires additional effort: a separate main class is needed in which Karate Main is called. This must be located in the “cucumber.api.cli” package.

4.2.5 API Testing with Karate

4.2.5.1 Matching Responses

- Karate offers extensive functions for validating API responses, especially JSON and XML data. With the Gherkin syntax, expected data structures can be defined directly in the test. Karate then automatically checks whether the actual response matches the expected one. Validation errors are clearly displayed in the integrated report, including the exact step in which the error occurred. This matching is initiated by the keyword “match”, followed by which response, e.g., response[0], and which element should correspond to the expected value. It is also possible to check all elements contained in a response. This is done using the == and the subsequent content specification.

4.2.5.2 Variables

- In Karate, variables can be easily set and used within Gherkin tests. These variables allow dynamic values (e.g., tokens, IDs) to be stored for later reuse. These values can then be reused in later requests or validations. Variables are flexible and can be used both locally within a test and globally in the test run. A variable is introduced with * def variable name. This is followed by its content. To use it, the following syntax is required: ‘#(variable name)’. Entire JSON objects can also be stored in multi-line variables; for this, """ is required to open and """ to close.

4.2.5.3 POST methods

- Karate supports the direct execution of HTTP POST requests through special keywords in the Gherkin syntax. The request body (e.g., JSON) is definable and can be sent directly to the API. Karate automatically validates the response and then processes it. For POST requests, it is sufficient to specify method post in Gherkin.

4.2.5.4 PUT methods

- Similar to POST requests, Karate allows PUT requests to be sent with a defined body (e.g., JSON). The response can then be checked to ensure that the update has been made. As with POST, the transfer method must be specified here using method get.

4.2.5.5 DELETE methods

- Karate also supports DELETE requests to remove resources. These requests can be executed without a body. Additional parameters can be transferred optionally. Karate checks the response for success or error, and the integrated validation ensures that the deletion operations were successful. In the Gherkin scenario, the entry to be deleted must be specified here and the method set to delete.

4.2.5.6 Tags

- In Karate, tests can be tagged to group them specifically or to execute only certain tests. This is comparable to the tags in TestNG or JUnit. These tags are defined directly in the Gherkin feature files using @TagName before the scenarios or features. This allows flexible control of test runs, e.g., to execute only regression tests, smoke tests, or special scenarios. It is also possible to write multiple tags to a scenario. This ensures simple and efficient test management.

4.2.5.7 Loading external scenarios

- Karate also allows you to load external scenarios or feature files. This makes it easy to reuse test cases and improves organization. These are integrated into a project using the command read(‘filename’). If necessary, individual or multiple external scenarios can be incorporated into a test run.

4.3 Data-Driven Testing with Karate

- Karate supports data-driven testing. These tests can be performed with different input data. Multiple data sets are defined in a table or in external files (e.g., CSV, JSON). Similar to the Cucumber data tables mentioned above, these run automatically and iteratively one after the other.

The familiar structure of scenario outlines can also be used in Karate. Alternatively, external data sources can be integrated. This enables the dynamic integration of different parameters into API requests. This significantly increases test coverage and reusability, as different scenarios can be tested with minimal effort.

4.4 Using Hooks

Hooks in Karate are special functions or instructions that are executed before or after certain test phases. Similar to Cucumber, recurring tasks can be automated, or the test sequence can be controlled.

Hooks in Karate:

Karate.callSingle('classpath:.../my.feature'; is stored in karate.config.js to execute a scenario once before a scenario

Background to perform preparations for all scenarios once before the first step

* configure to execute steps once after a scenario

4.5 Integration of Mocking Functions

Mocks are simulated objects used in software testing to isolate dependencies. They imitate the behavior of real components and enable controlled testing. In general, a distinction is made between different types of mocks. For example, so-called fakes only contain the basic implementation of an object with severely limited functionality. For this syllabus, the term mock is generally used to refer to a simulated object that ensures the necessary test functionality.

4.5.1 Advantages

- Independence from other parts:

Mocks isolate the component under test so that tests are not influenced by other parts of the system.

- Allows early testing:

With mocks, components can be tested early in the development phase, before all the components needed for a test are ready. Unfinished parts are replaced by mocks.

- Provoke rare situations:

Mocks enable the simulation of unusual or difficult-to-achieve scenarios that rarely occur in the real system.

- Saves resources:

The use of mocks eliminates the need for complex setups and real systems, which reduces time and costs.

4.5.2 Karate Mocking

- Karate also supports mocking API endpoints. For this purpose, Karate provides its own mock server, which can be managed in Java via lifecycle management.
- Existing functionality allows different requests and responses to be predefined. This means that tests can be run against this mock server. Request parameters can also be defined, which makes API testing very flexible.

The mock server must be called within a test. The path for the feature file can then be transferred via the builder class and the test can be executed. The test case can then be checked with the appropriate assertion. It is advisable to test for FailCount in order to exclude failed requests. getFailCount can be used to check whether errors have occurred.

4.6 Best Practices, Optimizations & Pros/Cons

4.6.1 Best Practices

To use mocks effectively in software testing, they should only be used when necessary to avoid unnecessary complexity. It is important not to make the mocks too complex so that they remain clear and maintainable. In addition, mocks should not influence each other in order to maintain the reliability and independence of tests. Regular refactoring of mocks helps to keep them up to date and clear. Finally, it is advisable to also test the real object to ensure that the integration works smoothly and no unexpected problems arise. Unforeseen issues can occur with the real object despite careful mock testing.

4.6.2 Mocks & Test Speed

Mocks contribute significantly to increasing test speed, as they are usually faster than real systems. They require less setup time, which simplifies and speeds up test preparation. The actual test execution with mocks is significantly shorter, as no complex processes or external resources need to be involved. Nevertheless, it is important to also perform tests with real objects to check the actual integration and interaction in the system. A balanced combination of mock tests and real system tests ensures efficient development cycles and reliable results.

4.6.3 Mock Optimizations

To use mocks efficiently, they should be designed to be reusable as far as possible. This avoids redundancy and increases maintainability. Profiling and monitoring mocks helps to monitor their performance and identify potential bottlenecks at an early stage. It is important to regularly review the use of mocks to ensure that they are always functioning optimally. To conserve resources, mocks should only be loaded when there is an actual need. This generally also shortens test runtimes. These optimizations help to make the test environment more stable and efficient.

4.6.4 Advantages and Disadvantages of Mocks

- Advantages of mocks:

The use of mocks enables very early testing, even before all components have been fully developed. This allows errors and misunderstandings to be detected and corrected at an early stage. Mocks offer high test speed because they enable quick and easy simulations of real components. They allow isolated testing of individual units, which facilitates debugging. In addition, edge cases are easier to test. Special scenarios can be simulated in a targeted manner. Overall, mocks simplify the testing of complex systems and promote efficient development.

- Disadvantages of mocks:

However, mock objects themselves can become very complex, especially with extensive or dynamic APIs. This leads to high maintenance costs, as changes to the API also affect the mock implementations and must be adjusted regularly. There is also a risk that potential errors will remain undetected due to abstraction in the code. Tests with mocks do not always correctly reflect all real interactions. This can lead to a false sense of security or unexpected problems in the real system.

Chapter 5 – CI/CD with Cucumber

Key Terms

Continuous Integration (CI), Continuous Deployment (CD)

LO-5.1	K1	Understand the basics of CI/CD and recognize their significance and benefits in software development
LO-2.2	K3	Set up a simple CI/CD pipeline with Jenkins. Practical skills for pipeline setup are necessary to apply theoretical knowledge.
LO-5.3	K2	Understand the BDD workflow and its integration into Jenkins
LO-5.4	K2	Generate and interpret Cucumber test reports. Reports provide feedback for the team and help identify issues in the development process
LO-5.5	K1	Understand the importance of structured test organization within a CI/CD process. Clear organization helps increase efficiency and maintainability

5.1 Continuous Integration / Continuous Deployment

5.1.1 Continuous Integration (CI)

Continuous integration (CI) is a proven practice in software development. It involves regularly integrating code changes – often several times a day – into a shared repository. The aim is to improve the quality of the software and make the development process more efficient.

Key Aspects of CI:

- Integration of new components

With CI, new components or functions are continuously integrated into the existing system. Developers thus merge their changes early and often. This allows conflicts and integration problems to be identified at an early stage. This makes it much easier to add new components, as they can be tested and integrated step by step.

- Improving software quality through testing

A key advantage of CI is the automatic testing of the code with every commit. After the code commit, the change goes through several stages: build, unit tests, and integration tests. These automated tests ensure that new changes do not affect existing functions and that the software remains stable.

- Simplification of adding new components

Automating the testing and integration processes simplifies the addition of new components. Developers can quickly receive feedback on whether their changes work and do not cause errors. This speeds up the development cycle and promotes an agile way of working.

5.1.2 Continuous Deployment (CD)

Continuous deployment (CD) is a process in software development in which software changes are automatically delivered directly to the production environment. After a successful continuous integration (CI) process, the tested components are automatically transferred to production, so that new features and updates are available to users quickly and reliably.

CD Process Steps:

1. Software delivery process after CI

The CD process builds on CI and generally involves several steps to ensure that only high-quality software enters production.

2. Review:

Before a version is deployed to the staging environment, the code is reviewed by developers or automated systems. This involves checking code quality, security aspects, and functionality, for example.

3. Staging:

After a successful review, the software is deployed in a staging environment. This environment simulates the production environment as closely as possible. Final tests are carried out here to ensure that everything works smoothly and no unexpected problems arise.

4. Production:

After all tests and reviews have been successfully completed, the software is rolled out automatically or manually into the production environment. Users thus quickly benefit from new features and improvements.

The processes mentioned in CI and CD are only the basis and can be expanded with additional processes such as reporting.

5.2 Jenkins Build Server

[JE1]

The Jenkins build server is an open-source automation platform that is primarily used in the areas of continuous integration (CI) and continuous delivery (CD). It enables software projects to be built, tested, and deployed automatically. This makes the development process more efficient. Jenkins supports extensive reporting functions to transparently display the status and quality of builds. Configuration is flexible via so-called Jenkins files. In these files, pipelines and automation processes are versioned and defined in a traceable manner. This makes Jenkins a powerful solution for automated software development, CI, and CD.

5.3 BDD Workflow with Jenkins

- In the Jenkins interface provided, a build process can be created via the “New Item” tab. The “Build now” field can be used to trigger an existing build process. In addition, the queue shows which jobs are currently active. Existing reports can also be displayed within the Jenkins interface.
- Jenkins offers the option of integrating Maven projects and then executing them. The start condition within the build configuration can be used to set when a build starts and which options are taken into account. Both the Maven POM and the corresponding Maven goals are configurable.
- In the post-build actions, it is possible to specify the settings for report generation, such as a Cucumber report.

In general, Jenkins calls the JUnit or TestNG Runner, which in turn executes the feature files. After this execution, the test reports are created. Since these reports are usually stored as JSON and HTML, they can also be conveniently transferred to other systems or people.

5.4 Reporting and Test Reports

A report JSON is required for test reporting and must be set by the corresponding runner. It may also be necessary to customize the runner class. In some cases, it may also be necessary to customize the Maven Pom to trigger the corresponding creation process. This JSON can then be used to create reports in HTML or other formats.

5.5 Best Practices for Test Organization in CI/CD

- Notifications:

Critical errors should trigger an automatic notification. For example, sending notifications via email or Slack is particularly suitable here and can be automatically integrated into the build pipeline. This allows teams to respond quickly to problems and maintain high software quality.

- Error handling:

It is advisable to implement meaningful logging within the pipelines. Detailed logs help to quickly identify the causes of errors and perform root cause analysis efficiently.

- Avoidance of large shared libraries:

It is advisable to minimize extensive shared libraries or dependencies. Instead, components should be kept modular and only the necessary parts should be used in the respective pipelines. This increases flexibility, reduces complexity, and minimizes potential conflicts.

References

Specific references

Reference	Source
[BT1]	A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives - By L. Anderson, P. W. Airasian, and D. R. Krathwohl (Allyn & Bacon 2001)
[BT2]	https://www.apu.edu/live_data/files/333/blooms_taxonomy_action_verbs.pdf
[MS1]	Softwaretesting kompakt: Grundlagen von Tests und Testautomatisierung mit Java (IT kompakt) – By Pascal Moll and Daniel Sonnet (Springer Vieweg Wiesbaden, 2025)
[CC1]	https://cucumber.io/docs
[KA1]	https://www.karatelabs.io/
[JE1]	https://www.jenkins.io/
[AW1]	https://aws.amazon.com/what-is/api/
[TE1]	https://testng.org/