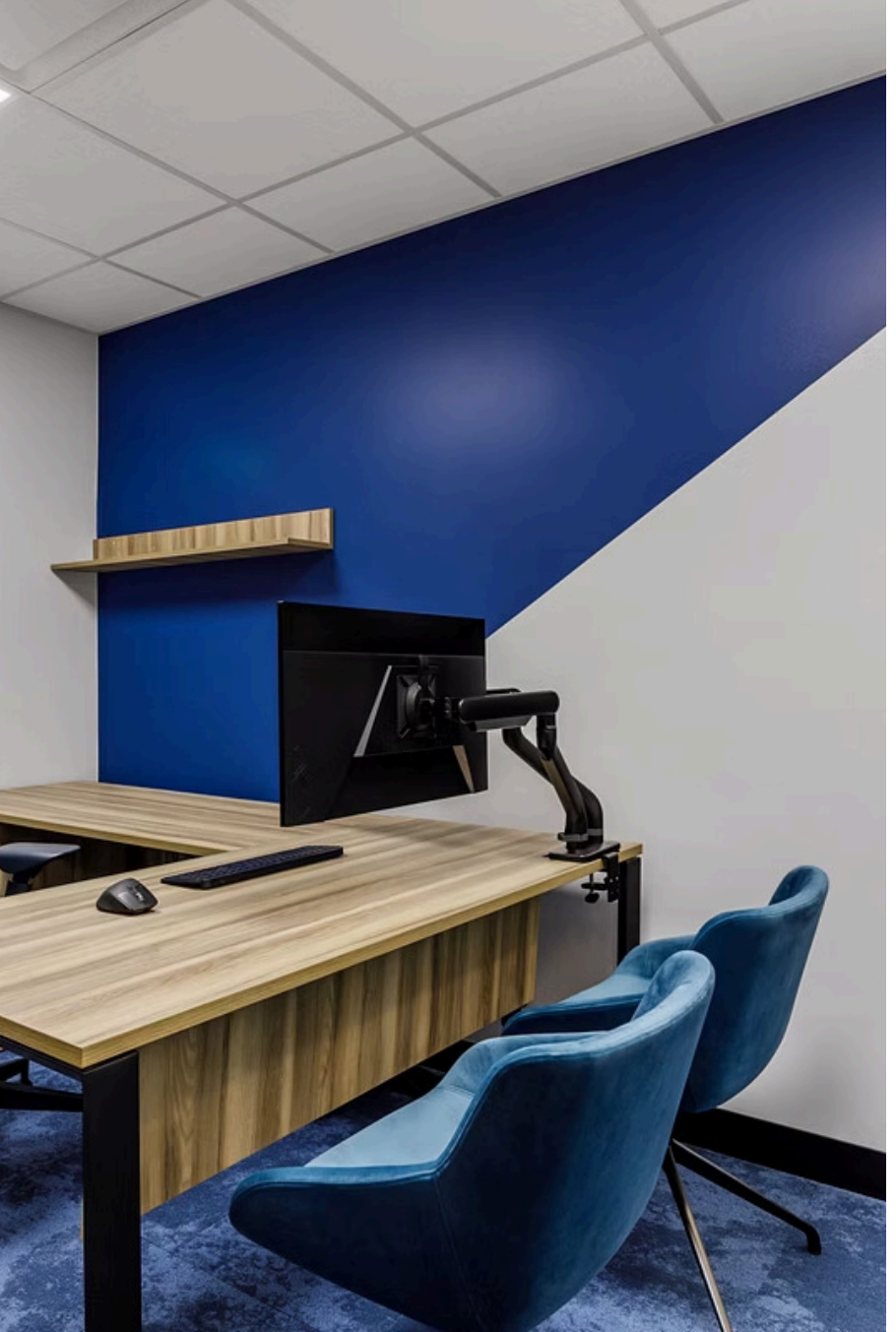


Behavior-Driven Development (BDD) & Model-Based Testing

Dependencies, Flows, and State Modelling

PROFESSIONAL TRAINING

QA ENGINEERING

What is Behavior-Driven Development (BDD)?

Behavior-Driven Development (BDD) is a collaborative software development approach that extends Test-Driven Development (TDD) by emphasising communication between developers, QA engineers, and business stakeholders. Rather than focusing solely on technical tests, BDD centres on defining the expected behaviour of a system using natural language that all team members can understand.

BDD bridges the gap between business requirements and technical implementation. By expressing requirements as concrete examples of system behaviour, teams create a shared understanding that reduces ambiguity and ensures everyone is aligned on what needs to be built and tested.

At its core, BDD promotes writing specifications in a structured, human-readable format before any code is written, ensuring that testing becomes an integral part of the development process rather than an afterthought.

Why BDD is Used in Modern Agile and QE Practices

Alignment with Agile Principles

BDD supports iterative development by enabling quick feedback loops. Scenarios written in plain language can be reviewed during sprint planning, ensuring that acceptance criteria are clear before development begins. This approach reduces rework and accelerates delivery cycles.

Living Documentation

BDD scenarios serve as executable specifications that remain current throughout the project lifecycle. Unlike traditional documentation that quickly becomes outdated, BDD scenarios are continuously validated through automation, providing reliable reference material for the entire team.

Enhanced Collaboration

BDD facilitates three-amigos sessions where business analysts, developers, and testers collaborate to define behaviour, eliminating silos and ensuring shared ownership of quality.

Early Defect Detection

By defining expected behaviour upfront, teams identify gaps and inconsistencies in requirements before coding begins, significantly reducing the cost of defect resolution.

Test Coverage Transparency

Stakeholders can easily understand what has been tested and what remains, improving visibility into quality and risk across the project.

BDD vs Traditional Testing Approaches

Understanding the fundamental differences between BDD and traditional testing methodologies helps teams recognise when and why to adopt BDD practices. The comparison below highlights how BDD transforms the testing mindset from verification to specification.

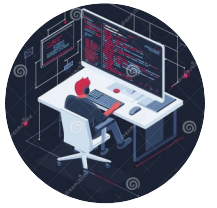
Aspect	Traditional Testing	Behavior-Driven Development
Focus	Verifying technical functionality against specifications	Defining and validating expected system behaviour
Language	Technical terminology, test case IDs, technical steps	Business-readable scenarios using natural language
When Written	After development is complete or in progress	Before development begins, during requirements analysis
Collaboration	Primarily QA-driven with limited stakeholder input	Collaborative effort involving business, development, and QA
Perspective	System-centric, focusing on what the system does	User-centric, focusing on why users need the behaviour
Documentation	Separate test cases that may become outdated	Living documentation that evolves with the system

Role of Collaboration in BDD

BDD's power lies in its collaborative approach, bringing together three key perspectives: business, development, and quality assurance. This collaboration occurs throughout the entire development lifecycle, from initial requirement discussions to scenario refinement and execution.

Business Stakeholders

Define the desired behaviour based on user needs and business goals. They provide domain knowledge, clarify acceptance criteria, and validate that scenarios accurately reflect business requirements.



Development Team

Translate behaviour scenarios into technical implementation. They identify technical constraints, suggest alternative approaches, and ensure scenarios are testable and implementable.

Quality Assurance

Facilitate scenario creation, ensure comprehensive coverage, identify edge cases, and automate scenario execution. QA acts as a bridge between business intent and technical implementation.

This three-amigos approach ensures that scenarios are simultaneously relevant to the business, technically feasible, and thoroughly testable, creating a robust foundation for quality delivery.

Benefits of BDD: Shared Understanding

Reduced Ambiguity in Requirements

Concrete examples eliminate interpretation gaps. When a requirement states "users should be able to log in," a BDD scenario specifies exact conditions, actions, and outcomes, leaving no room for misunderstanding.

Faster Onboarding

New team members can quickly understand system behaviour by reading scenarios. The business-readable format requires minimal technical knowledge, accelerating knowledge transfer and reducing dependency on tribal knowledge.

Improved Estimation Accuracy

Clearly defined scenarios enable more accurate effort estimation. Development and testing teams can assess complexity based on specific behaviours rather than vague requirements, leading to better sprint planning.

Enhanced Requirement Traceability

Each scenario links directly to a business requirement and can be traced through to implementation and test execution. This traceability improves compliance, audit readiness, and impact analysis when requirements change.

BDD Interview Keywords and Concepts

Mastering BDD requires familiarity with key terminology frequently assessed in interviews and essential for effective communication in Agile teams. Understanding these concepts demonstrates proficiency in modern quality engineering practices.

Behaviour

The observable outcomes and responses of a system from a user's perspective, rather than internal technical implementation details.

Collaboration

The three-amigos approach involving business, development, and QA working together to define and validate system behaviour.

Acceptance Criteria

Specific conditions that must be satisfied for a feature to be considered complete, expressed as executable scenarios.

Living Documentation

Specifications that remain current through continuous execution, serving as both requirements and automated tests.

Specification by Example

Defining requirements using concrete examples of system behaviour rather than abstract statements.

Ubiquitous Language

A shared vocabulary used consistently across business and technical teams to describe domain concepts.

BDD Knowledge Check: Multiple Choice Questions

Test your understanding of BDD fundamentals with these interview-style questions. Consider each option carefully before selecting your answer.

Question 1: Which statement best describes the primary goal of Behavior-Driven Development?

- A) To automate all test cases using Selenium WebDriver
- B) To create a shared understanding of system behaviour between business, development, and QA
- C) To replace manual testing with automated testing
- D) To write unit tests before writing production code

Correct Answer: B – BDD focuses on collaboration and shared understanding through behaviour specification.

Question 2: In a BDD workflow, when should scenarios ideally be written?

- A) After development is complete, during the testing phase
- B) During the sprint retrospective
- C) Before development begins, during requirement refinement
- D) After user acceptance testing has identified defects

Correct Answer: C – BDD scenarios are specifications written before development to guide implementation.

GHERKIN LANGUAGE

What is Gherkin?

Gherkin is a business-readable, domain-specific language used to describe software behaviour without detailing implementation. It serves as the syntax for writing BDD scenarios, using a structured format that both technical and non-technical stakeholders can understand and contribute to.

Gherkin follows a simple keyword-driven structure that organises scenarios into logical blocks. Each scenario describes a specific behaviour through a series of steps representing preconditions, actions, and expected outcomes. This structure enforces clarity and consistency across all behaviour specifications.

Written in plain text, Gherkin files typically use the .feature extension and can be version-controlled alongside source code. This approach ensures that behaviour specifications evolve in tandem with the application, maintaining alignment between documentation and implementation throughout the development lifecycle.

Purpose of Gherkin in BDD

1 Bridge Communication Gaps

Provides a common language that business analysts, developers, and testers can all read and write without requiring technical programming knowledge.

2 Enable Automation

Gherkin scenarios can be parsed by BDD frameworks and mapped to executable code, creating a direct link between requirements and automated tests.

3 Document Behaviour

Serves as living documentation that describes what the system does from a user perspective, remaining current through continuous execution.

Gherkin Keywords: Building Blocks of Scenarios

Gherkin uses a small set of keywords to structure scenarios. Each keyword serves a specific purpose in describing behaviour, and understanding their proper usage is essential for writing effective BDD scenarios.

Feature

Describes a high-level functionality or capability of the system. Each .feature file begins with this keyword followed by a brief description of the feature being tested.

Example: Feature: User Authentication

Scenario

Represents a specific example of behaviour within a feature. Each scenario should test one particular aspect or path through the functionality.

Example: Scenario: Successful login with valid credentials

Given

Establishes the preconditions or context before an action occurs. Sets up the initial state of the system required for the test.

Example: Given the user is on the login page

When

Describes the action or event that triggers the behaviour being tested. Should represent a single, clear action performed by the user or system.

Example: When the user enters valid credentials and clicks login

Then

Specifies the expected outcome or result after the action. Defines what should happen if the system behaves correctly.

Example: Then the user should be redirected to the dashboard

And / But

Continues the previous step type to add additional context, actions, or expectations without repeating Given, When, or Then keywords.

Example: And the welcome message should be displayed

Scenario Readability: The Foundation of BDD

Readable scenarios are the cornerstone of effective BDD implementation. A well-written scenario should be understandable by anyone on the team, regardless of their technical background. Readability ensures that scenarios serve their purpose as shared specifications and communication tools.

Each scenario should tell a clear story with a beginning (Given), middle (When), and end (Then). Avoid technical jargon, implementation details, or references to UI elements like button IDs. Instead, focus on business-level actions and outcomes that describe what the user wants to accomplish and why.

Keep scenarios concise and focused on a single behaviour. If a scenario becomes too long or complex, it likely covers multiple behaviours and should be split into separate scenarios. Each scenario should be independently understandable without requiring readers to reference other scenarios or external documentation.

Mapping Business Requirements to Gherkin Tests

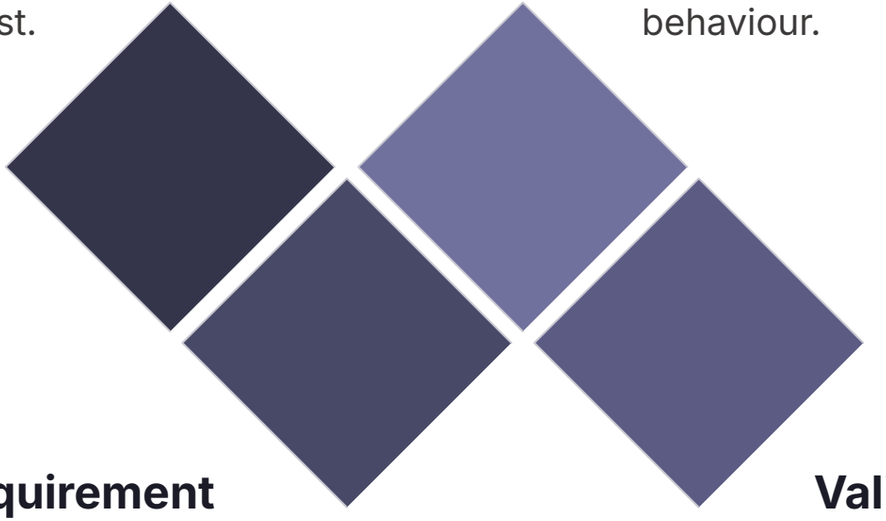
Translating business requirements into Gherkin scenarios requires a systematic approach that ensures complete coverage whilst maintaining clarity. The process involves identifying distinct behaviours within each requirement and expressing them as concrete examples.

Identify Behaviours

Extract distinct behaviours and rules to test.

Write Examples

Draft clear Gherkin scenarios for each behaviour.



Analyse Requirement

Review business needs and acceptance criteria.

Validate with Stakeholders

Confirm scenarios reflect expectations and coverage.

Begin by analysing the business requirement to identify all possible user journeys and edge cases. For each distinct behaviour, create a separate scenario that illustrates how the system should respond. Consider both happy paths and alternative flows, including error conditions and boundary cases. Validate completed scenarios with business stakeholders to ensure they accurately represent intended behaviour before proceeding to implementation.

Simple Gherkin Scenario Example

This example demonstrates a complete Gherkin scenario for an e-commerce checkout process. Notice how the scenario reads like a natural conversation whilst providing precise, testable specifications.

```
Feature: Shopping Cart Checkout

  Scenario: Complete purchase with valid payment details
    Given the user has added items worth £150 to the cart
    And the user is on the checkout page
    When the user enters valid delivery details
    And the user enters valid payment card information
    And the user clicks "Complete Purchase"
    Then the order should be confirmed
    And the user should receive an order confirmation email
    And the inventory should be updated to reflect the purchase
```

This scenario clearly establishes context (Given), describes actions (When), and specifies expected outcomes (Then). Notice how it avoids technical details like API endpoints or database tables, focusing instead on observable behaviour from a business perspective.

Good vs Poor Gherkin Practices

Understanding the distinction between effective and ineffective Gherkin scenarios is crucial for maintaining quality BDD specifications. The comparison below highlights common pitfalls and best practices.

❌ Poor Practice

```
Scenario: Test login
  Given I open browser
  And I navigate to
    "https://app.example.com"
  And I find element by id
    "username"
  When I type "john@example.com"
  And I find element by id
    "password"
  And I type "Pass123"
  And I click button id
    "submitBtn"
  Then I should see dashboard
```

Issues: Too technical, exposes implementation details, focuses on UI elements rather than behaviour, difficult for non-technical stakeholders to understand.

✅ Good Practice

```
Scenario: User logs in with
  valid credentials
  Given the user is on the
    login page
  When the user logs in with
    valid credentials
  Then the user should see
    their personalised dashboard
  And the user's name should
    be displayed in the header
```

Strengths: Business-readable, focuses on behaviour not implementation, clearly describes user intent and expected outcomes, maintainable when UI changes.

Gherkin Best Practices: Additional Guidelines

Use Active Voice

Write steps from the user's perspective using active voice. Say "the user enters their email" rather than "email is entered".

One Scenario, One Behaviour

Each scenario should test a single specific behaviour. Avoid combining multiple unrelated behaviours in one scenario.

Declarative Over Imperative

Describe what should happen, not how to make it happen. Focus on business intent rather than step-by-step instructions.

Avoid Scenario Dependencies

Each scenario should be independently executable. Never assume another scenario has run first or created specific data.

Use Background Sparingly

Background steps run before every scenario in a feature. Only use for truly common preconditions to avoid repetition.

Maintain Consistent Language

Use the same terms throughout your scenarios as the business uses. This ubiquitous language prevents confusion and misinterpretation.

Gherkin Knowledge Check: Interview Questions

Question 1: Which Gherkin keyword should be used to describe the action that triggers the behaviour being tested?

- A) Given – it sets up the initial context
- B) When – it describes the action or event
- C) Then – it specifies the expected outcome
- D) And – it continues the previous step

Correct Answer: B – When describes the action that triggers the behaviour under test.

Question 2: What is the main problem with the following Gherkin step: "Given I click on element with xpath '//div[@id='loginBtn']'"?

- A) XPath selectors should use single quotes instead of double quotes
- B) It exposes technical implementation details rather than describing behaviour
- C) The Given keyword should be When since it's an action
- D) Element IDs should be specified using CSS selectors not XPath

Correct Answer: B – Gherkin should describe business behaviour, not technical implementation.

Question 3: A common interview trap: When should you use Given vs When in a Gherkin scenario?

Answer: Use Given for preconditions and context (system state before the test), and When for the action that triggers the behaviour being tested. A common mistake is using When for setup steps or Given for actions.

BDD EXECUTION

How BDD Scenarios Are Executed

BDD execution transforms human-readable Gherkin scenarios into automated tests through a mapping layer that connects each step to executable code. This process enables scenarios to serve dual purposes: as specifications for human understanding and as automated tests for continuous validation.

When a BDD framework executes a scenario, it parses each Gherkin step and matches it to corresponding code implementations called step definitions. These step definitions contain the actual automation logic—interacting with the application, performing actions, and verifying outcomes. The framework orchestrates the execution sequence, running each step in order and reporting results.

This architecture separates behaviour specification from implementation details. Business stakeholders write and maintain scenarios in Gherkin whilst developers and test engineers implement and maintain step definitions. This separation enables scenarios to remain stable even when underlying implementation or UI elements change, significantly reducing maintenance overhead.

Mapping Gherkin Steps to Automation Code

The connection between Gherkin steps and executable code occurs through pattern matching. Each step definition uses regular expressions or string matching to identify which Gherkin steps it should handle. Understanding this mapping is essential for effective BDD implementation.

Gherkin Scenario

```
Scenario: Add item to cart
  Given the user is viewing
    the product catalogue
  When the user adds a
    laptop to the cart
  Then the cart should
    contain 1 item
  And the cart total should
    be £899
```

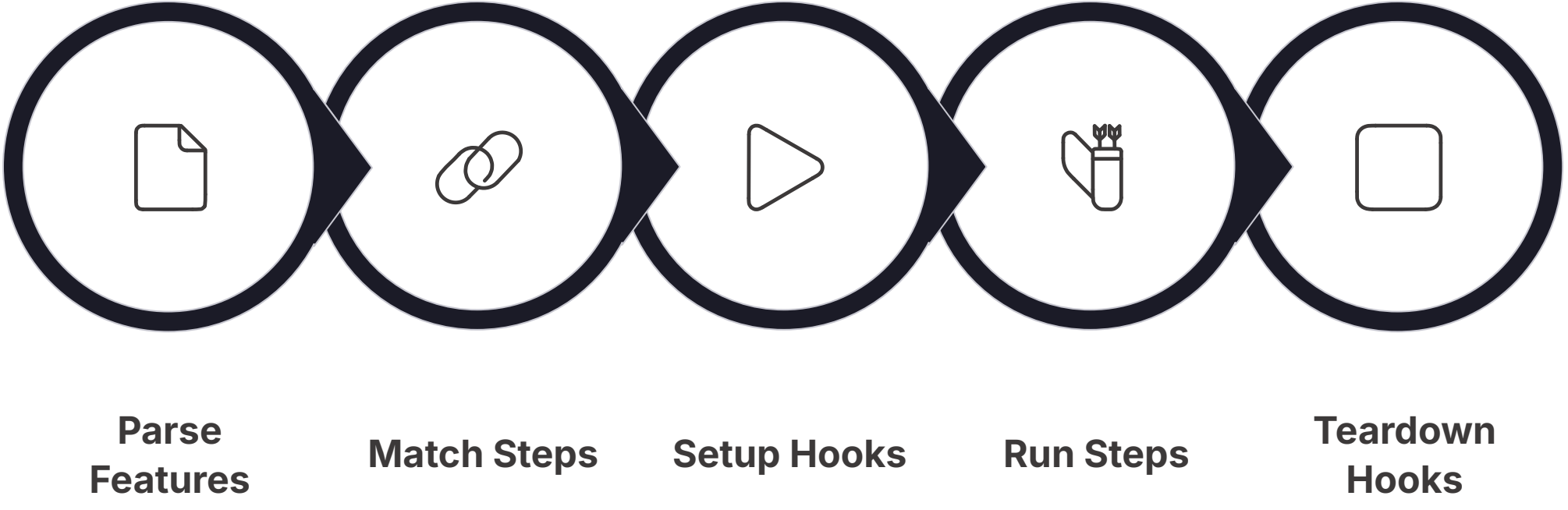
Step Definition Mapping (Conceptual)

```
@Given("user is viewing the
  product catalogue")
  -> Navigate to catalogue

@When("user adds a {product}
  to cart")
  -> Find product
  -> Click add to cart

@Then("cart should contain
  {int} item(s)")
  -> Verify cart count

@Then("cart total should be
  {currency}")
  -> Verify total amount
```

Parameters can be extracted from Gherkin steps and passed to step definitions, enabling reusable steps that work with different data values. This parameterisation reduces code duplication and makes scenarios more maintainable.

BDD Execution Flow

Understanding the complete execution flow helps teams troubleshoot issues and optimise test performance. The process involves several distinct phases from scenario parsing to result reporting.



The execution begins with the BDD framework parsing feature files to extract scenarios and steps. For each scenario, the framework matches Gherkin steps to their corresponding step definitions. Before executing the scenario, any setup hooks run to prepare the test environment. The framework then executes each step definition in sequence, stopping immediately if any step fails. After scenario completion, teardown hooks clean up test data and resources. Finally, the framework generates reports showing which scenarios passed or failed, including details of any failures for debugging purposes.

BDD Tools Overview: Cucumber Framework

Cucumber is the most widely adopted BDD framework, supporting multiple programming languages including Java, JavaScript, Ruby, and Python. It provides the infrastructure for parsing Gherkin syntax, matching steps to code, executing tests, and generating reports. Understanding Cucumber's role helps teams implement BDD effectively.

Framework Components

Cucumber consists of a Gherkin parser, step definition matcher, test runner, and report generator. These components work together to transform scenarios into executable tests whilst maintaining separation between specification and implementation.

Integration Capabilities

Cucumber integrates with popular testing tools and frameworks such as Selenium for web testing, REST Assured for API testing, and JUnit or TestNG for test execution. This flexibility enables teams to test across different application layers using consistent BDD approach.

Reporting and Analytics

Cucumber generates detailed execution reports showing pass/fail status, execution time, and failure details. Third-party plugins provide enhanced reporting with graphs, trends, and integration with test management systems for comprehensive test analytics.

BDD Execution Knowledge Check

Question 1: What is the primary purpose of step definitions in a BDD framework?

- A) To write the Gherkin scenarios in the feature files
- B) To map Gherkin steps to executable automation code
- C) To generate test reports after execution
- D) To parse feature files and validate Gherkin syntax

Correct Answer: B – Step definitions connect human-readable steps to executable code.

Question 2: When using Cucumber, if a single step in a scenario fails, what happens to the remaining steps?

- A) They continue executing to gather more failure information
- B) They are skipped and marked as pending
- C) They are retried automatically up to three times
- D) They execute but failures are not reported

Correct Answer: B – Cucumber skips remaining steps after a failure to avoid cascading errors.

DEPENDENCIES

What Are Dependencies in Software Systems?

In software systems, a dependency exists when one component relies upon another component to function correctly. Dependencies create relationships where changes, failures, or unavailability of one component directly impact other components. Understanding these relationships is fundamental to effective testing strategy and risk management.

Dependencies manifest at multiple levels within an application architecture. At the code level, modules and classes depend on each other. At the service level, microservices depend on other services and external APIs. At the infrastructure level, applications depend on databases, message queues, and third-party platforms. Each dependency introduces potential points of failure and complexity that must be managed during testing.

Modern distributed systems amplify dependency challenges. A single user transaction might traverse dozens of services, each with its own dependencies. This complexity makes dependency identification and management critical skills for quality engineers working in contemporary software development environments.

Why Dependency Identification Is Critical in Testing

- **Risk Assessment**

  Identifying dependencies reveals potential failure points. Understanding which components depend on external services helps prioritise testing efforts and implement appropriate fallback mechanisms.

- **Test Planning**

  Dependency maps inform test environment setup and data requirements. Knowing dependencies upfront prevents test blockages caused by missing prerequisites or unavailable services.

- **Impact Analysis**

  When changes occur in one component, dependency knowledge enables accurate assessment of which other components require regression testing to ensure stability.

Real-World Impact

A 2023 industry survey found that 60% of production incidents in microservices architectures stemmed from unmanaged dependencies. Teams that maintained explicit dependency documentation experienced 40% fewer critical defects and resolved issues 3x faster than teams without dependency tracking.

Unidentified dependencies often surface as cryptic failures in production, requiring extensive investigation to diagnose. Proactive dependency identification shifts this discovery left, enabling teams to design tests that validate dependency behaviour before deployment.

Types of Dependencies: Functional Dependencies

Functional dependencies exist when one feature or module requires another feature to provide its expected functionality. These dependencies form the logical flow of business processes and represent how different parts of the application work together to deliver value.

Example: In an e-commerce application, the checkout process has a functional dependency on the shopping cart. Users cannot complete a purchase without first adding items to their cart. Similarly, order confirmation depends on payment processing, which in turn depends on payment gateway integration.

Testing functional dependencies requires validating that dependent features fail gracefully when their dependencies are unavailable. For instance, if the payment gateway is down, the checkout process should display an appropriate error message rather than allow users to proceed with incomplete transactions.

Types of Dependencies: Data Dependencies

Database Dependencies

Application components depend on specific data structures, relationships, and constraints in databases. Changes to schema affect all dependent components, requiring coordinated testing across layers.

Data Transformation

Processes that transform data from one format to another create dependencies. Downstream systems rely on consistent data formats and structures from upstream transformation processes.

Master Data Dependencies

Multiple systems often share master data like customer information or product catalogues. Changes to master data impact all dependent systems, requiring synchronised testing to ensure consistency.

Data dependencies require careful test data management. Tests must ensure that necessary data exists in the correct state before execution and that test data in one area doesn't inadvertently affect tests in another area through shared data stores.

Types of Dependencies: Environment and External Dependencies

Environment dependencies relate to infrastructure and external system dependencies involve third-party services beyond the organisation's direct control. Both types significantly impact testing strategy and require specific management approaches.

Dependency Type	Description	Testing Considerations
Configuration	Application behaviour depends on environment-specific settings and feature flags	Test across configurations; validate fallbacks when config unavailable
Infrastructure	Dependencies on servers, networks, load balancers, and cloud services	Consider infrastructure variability; test failover and redundancy mechanisms
Third-Party APIs	Integration with external service providers like payment gateways or shipping APIs	Cannot control availability or performance; requires mocking or service virtualisation
Authentication Services	OAuth providers, LDAP, or single sign-on systems	Test both success and failure scenarios; validate session management
External Databases	Shared databases or data warehouses managed by other teams	Coordinate test data setup; avoid interfering with production or other teams' tests

Real-World Example: Payment Gateway Dependency

Consider an online retail platform's dependency on a payment gateway. This scenario illustrates how dependencies create testing challenges and require strategic management approaches.

Dependency Characteristics

- External service outside team control
- Critical path dependency—checkout cannot complete without it
- Variable response times affecting user experience
- Potential for service downtime or rate limiting
- Costs associated with transaction testing
- Sensitive data requiring secure handling
- Multiple failure scenarios to validate

Testing Implications

Teams cannot rely on the live payment gateway for automated testing due to costs, rate limits, and unpredictability. Instead, they must implement alternatives:

- Use payment gateway sandbox environments for integration testing
- Implement mocks for unit and component testing
- Create stubs simulating various response scenarios (success, decline, timeout, error)
- Test fallback behaviour when gateway unavailable
- Validate error handling for partial failures
- Perform limited end-to-end tests in production-like environments

Dependencies Interview Keywords

Dependency

A relationship where one component requires another to function, creating potential points of failure and testing complexity.

Coupling

The degree of interdependence between components. Tight coupling increases risk; loose coupling improves testability and resilience.

Integration Risk

The probability that interface mismatches or dependency failures will cause defects when components are integrated.

Dependency Inversion

Design principle where high-level modules don't depend on low-level modules; both depend on abstractions, improving testability.

Transitive Dependency

Indirect dependency where A depends on B, and B depends on C, meaning A implicitly depends on C.

Dependency Graph

Visual representation showing relationships between components, used for impact analysis and test planning.

Dependency Identification Knowledge Check

Question 1: Which type of dependency exists when the order confirmation feature cannot function without the payment processing feature completing successfully?

- A) Data dependency
- B) Functional dependency
- C) Environment dependency
- D) Configuration dependency

Correct Answer: B – This is a functional dependency where one feature requires another to provide its functionality.

Question 2: Your application depends on a third-party API that has rate limits and charges per transaction. What is the most appropriate testing strategy?

- A) Run all automated tests against the live API to ensure accuracy
- B) Use service virtualisation or mocking for most tests, with limited live API tests
- C) Avoid testing the integration entirely since it's a third-party service
- D) Only test the integration manually before each release

Correct Answer: B – Balance thorough testing with practical constraints using virtualisation for most tests.

MANAGING DEPENDENCIES

Challenges Caused by Unmanaged Dependencies

When dependencies remain unidentified or poorly managed, they create significant obstacles throughout the development lifecycle. These challenges manifest as delayed testing, increased defect rates, and reduced team productivity. Understanding these impacts motivates investment in proper dependency management practices.

Test Blockages

Tests fail or cannot execute because required dependencies are unavailable, misconfigured, or in an unexpected state. Teams waste time investigating environmental issues rather than finding product defects.

Flaky Tests

Tests produce inconsistent results due to timing issues, race conditions, or transient dependency failures. Flaky tests erode confidence in test automation and create debugging overhead.

Production Incidents

Defects escape to production because tests don't adequately cover dependency failure scenarios. Real users experience errors that could have been prevented with better dependency testing.

Maintenance Burden

Changes to shared dependencies require coordinated updates across multiple test suites. Without clear dependency documentation, teams struggle to identify which tests require updates.

Dependency Management Strategies

Effective dependency management requires a multi-faceted approach that addresses both technical and organisational challenges. The strategies below provide a framework for teams to maintain control over complex dependency networks whilst enabling efficient testing.

Document Dependencies

Maintain explicit documentation of all dependencies including type, criticality, and ownership. Use dependency graphs to visualise relationships and identify high-risk areas.

Isolate Components

Design systems with loose coupling to minimise dependency impact. Use abstractions and interfaces to decouple components, enabling independent testing and deployment.

Virtualise Services

Implement mocks, stubs, and service virtualisation to simulate dependencies during testing. This approach eliminates dependency on external systems and enables controlled failure testing.

Monitor Dependencies

Continuously monitor dependency health in test and production environments. Implement alerts for dependency failures to enable rapid response and resolution.

Stubbing and Mocking: Core Concepts

Stubs

Stubs provide predefined responses to method calls during testing. They replace real dependencies with simplified implementations that return controlled data. Stubs enable testing of specific scenarios without requiring the actual dependency to be available or in a particular state.

Use cases: Testing how your code handles specific responses from external services, isolating units for focused testing, eliminating unpredictable behaviour from dependencies.

Mocks

Mocks go beyond stubs by verifying that specific interactions occurred. They record which methods were called, with what parameters, and how many times. Mocks enable verification that your code correctly uses its dependencies.

Use cases: Verifying that your code calls dependency methods correctly, ensuring proper parameter passing, validating interaction sequences and frequency.

Both approaches enable testing in isolation, but serve different purposes. Use stubs when you need controlled inputs and mocks when you need to verify outputs and interactions. Many testing frameworks provide both capabilities, allowing teams to choose the appropriate technique for each testing scenario.

Service Virtualisation: Advanced Dependency Management

Service virtualisation creates sophisticated simulations of dependent systems, capturing their behaviour, performance characteristics, and failure modes. Unlike simple mocks or stubs, virtualised services replicate complex scenarios including latency, throughput limitations, and realistic error conditions.

This approach proves particularly valuable when testing against third-party APIs, legacy systems, or services still under development. Service virtualisation enables parallel development and testing, removing bottlenecks caused by dependency availability. Teams can test edge cases and failure scenarios that would be difficult or impossible to reproduce with real dependencies.

Benefits

Eliminates dependency on external system availability, reduces testing costs associated with third-party services, enables testing of scenarios difficult to create with real systems, and accelerates testing by removing wait times and coordination overhead.

Implementation Considerations

Service virtualisation requires initial investment to capture and configure service behaviours. Virtualised services must be maintained to remain synchronised with real service changes. Teams need tools like WireMock, Mountebank, or commercial platforms to implement effective service virtualisation.

Test Sequencing and Environment Isolation

Strategic test sequencing and environment isolation prevent dependency-related test failures and enable reliable automated testing. These practices ensure tests execute in predictable conditions without interference from concurrent test execution or residual state.

Test Sequencing Strategies

- **Sequential execution:** Run tests one at a time when they share dependencies or modify shared state
- **Dependency ordering:** Execute tests in order based on dependency graph to ensure prerequisites are met
- **Parallel with isolation:** Run independent tests concurrently whilst serialising dependent tests
- **Data partitioning:** Assign unique test data to each parallel execution to prevent conflicts

Environment Isolation Approaches

- **Dedicated environments:** Separate test environments for different test types (unit, integration, E2E)
- **Containerisation:** Use Docker or similar technologies to create isolated, reproducible test environments
- **Database isolation:** Each test suite uses separate database instances or schemas to prevent data conflicts
- **Namespace isolation:** Isolate test resources using namespaces in cloud platforms and orchestration tools

Dependency Management Flow

A systematic approach to managing dependencies throughout the testing lifecycle ensures comprehensive coverage whilst maintaining efficiency. The following flow illustrates the key decision points and activities.

Begin by identifying all dependencies for the component under test. Assess each dependency's criticality and the risk it introduces. For non-critical dependencies or those with high variability, implement mocking or stubbing. For complex, critical dependencies, consider service virtualisation. Reserve real dependency usage for final integration validation. Continuously monitor dependency health and update your management approach as systems evolve.

Case Study: Test Blockage Due to Dependency Failure

A financial services company experienced a critical test environment outage that illustrates the importance of proper dependency management. This real-world scenario demonstrates how unmanaged dependencies can cascade into significant business impact.

Scenario

The QA team's automated regression suite depended on a shared authentication service managed by another team. Without prior notice, the authentication service team deployed changes that altered the API contract. The QA team's tests began failing immediately, blocking all testing for a major release scheduled in three days.

Day 1: Discovery

All 2,000+ automated tests fail with authentication errors. Team spends 6 hours investigating, initially suspecting their own code changes.

1

Day 2: Resolution Attempt

2

Root cause identified as authentication service changes. Teams coordinate to create hotfix, but deployment requires approvals and change control processes.

Day 3: Recovery

3

Authentication service rolled back. Tests resume, but release delayed by one week due to insufficient testing time.

Lessons and Prevention: Implement service virtualisation for the authentication dependency, establish cross-team communication protocols for shared service changes, create contract tests to detect breaking changes early, and maintain rollback capability for critical shared services.

Dependency Management Knowledge Check

Question 1: What is the primary difference between a stub and a mock in testing?

- A) Stubs are used for unit testing whilst mocks are used for integration testing
- B) Stubs provide predefined responses; mocks verify that specific interactions occurred
- C) Stubs are faster to execute than mocks
- D) Mocks require third-party tools whilst stubs can be written manually

Correct Answer: B – The key distinction is that mocks verify interactions whilst stubs simply provide responses.

Question 2: Your team needs to test integration with a third-party payment API that charges per transaction and has strict rate limits. Which approach is most appropriate?

- A) Execute all automated tests against the production API to ensure accuracy
- B) Implement service virtualisation for automated tests with periodic validation against the real API
- C) Skip automated testing of the integration and rely solely on manual testing
- D) Create a simple stub that always returns success responses

Correct Answer: B – Service virtualisation balances thorough testing with practical constraints, supplemented by real API validation.

MODEL-BASED TESTING

What is Model-Based Testing (MBT)?

Model-Based Testing (MBT) is a software testing approach that uses formal models to represent system behaviour, structure, or data flow. Rather than writing individual test cases manually, testers create models that capture how the system should behave under various conditions. Test cases are then derived systematically from these models, ensuring comprehensive coverage based on the model structure.

MBT shifts focus from example-based thinking to model-based thinking. Instead of asking "what examples should I test?", testers ask "what are all the possible states, transitions, and paths through the system?" This approach reveals test scenarios that might be overlooked in traditional example-based testing, particularly edge cases and complex interaction patterns.

Models serve multiple purposes: they document system behaviour in a visual, precise format; they enable automated test generation; and they facilitate gap analysis by revealing untested areas. Common MBT models include state transition diagrams, decision tables, control flow graphs, and data flow diagrams.

Why Models Are Used in Test Design

Systematic Coverage

Models enable systematic exploration of all possible scenarios. By visualising system behaviour, testers can apply coverage criteria (e.g., all states visited, all transitions executed) to ensure thorough testing.

Complexity Management

Complex systems with numerous conditions, states, and paths become manageable when represented visually. Models break down complexity into understandable components and relationships.

Maintainability

When requirements change, updating a model is more efficient than revising numerous individual test cases. Test cases regenerated from updated models automatically reflect the changes.

Communication Tool

Models provide a shared language for business analysts, developers, and testers to discuss system behaviour, reducing misunderstandings and aligning expectations.

Example-Based vs Model-Based Testing

Understanding the distinction between traditional example-based testing and model-based testing helps teams select the appropriate approach for different testing challenges. Each approach has strengths suited to particular contexts.

Aspect	Example-Based Testing	Model-Based Testing
Approach	Derive individual test cases from requirements and experience	Create formal models; systematically generate test cases from models
Coverage	Dependent on tester experience and imagination	Mathematically defined coverage based on model structure
Scalability	Becomes unwieldy with system complexity; many manual test cases	Scales better; model complexity grows slower than individual test cases
Upfront Effort	Low—can start writing tests immediately	Higher—requires model creation before test generation
Maintenance	High—each test case may need individual updates	Lower—update model once; regenerate affected tests
Best Suited For	Simple features, exploratory testing, usability testing	Complex business logic, state-dependent systems, regulatory compliance

Benefits of Model-Based Testing

Defect Prevention

Creating models forces rigorous analysis of requirements, often revealing ambiguities, contradictions, and gaps before coding begins.

Objective Coverage

Coverage metrics become measurable and objective (e.g., "all transitions covered" vs. subjective "sufficient testing").

Test Automation Synergy

Models naturally support test automation by providing structured input for test generation tools and frameworks.

Regulatory Compliance

Many regulated industries require traceability between requirements and tests; models provide auditable evidence of systematic testing.

Resource Efficiency

Initial model investment pays dividends through automated test generation, reducing long-term testing effort and increasing efficiency.

Knowledge Capture

Models document tester knowledge and system understanding, preserving intellectual capital and reducing risk when team members change.

Model-Based Testing Knowledge Check

Question 1: What is the primary advantage of model-based testing over example-based testing?

- A) Model-based testing requires less upfront effort to begin testing
- B) Model-based testing provides systematic coverage based on model structure
- C) Model-based testing eliminates the need for automation frameworks
- D) Model-based testing works better for simple features with few scenarios

Correct Answer: B – MBT's key advantage is systematic, comprehensive coverage derived from formal models.

Question 2: Which scenario is MOST appropriate for applying model-based testing techniques?

- A) Testing the visual layout of a landing page
- B) Testing a workflow with multiple states and complex business rules
- C) Conducting usability testing with end users
- D) Performing exploratory testing to discover unexpected behaviours

Correct Answer: B – Complex workflows with states and rules benefit most from MBT's systematic approach.

Model-Based Testing Interview Keywords

Model

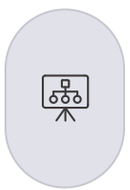
A formal representation of system behaviour, structure, or data that serves as the basis for systematic test case generation.

Abstraction

The process of simplifying complex systems by focusing on essential characteristics whilst omitting irrelevant details.

Coverage Criteria

Rules that define which elements of a model must be exercised by tests (e.g., all states, all transitions, all decision outcomes).



Test Generation

The process of automatically deriving test cases from models using algorithms and coverage criteria.

Test Oracle

The mechanism for determining whether test execution produced correct results, often derived from expected behaviours in the model.

Traceability

The ability to trace test cases back to specific model elements and forward to requirements, ensuring comprehensive coverage.

CONTROL FLOW

What is Control Flow in Application Logic?

Control flow describes the order in which individual statements, instructions, or function calls are executed within a program. It represents the logical flow of execution as the program runs, including how decisions are made and how loops are traversed. Understanding control flow is essential for designing tests that adequately exercise all execution paths through the code.

In most programming languages, control flow is determined by control structures such as if-else statements, switch cases, loops (for, while), and function calls. Each control structure creates branching points where execution can take different paths based on conditions or data values. Testing must account for these branches to ensure all logical paths execute correctly.

From a testing perspective, control flow analysis reveals the possible execution paths through a piece of code or a business process. Testers use control flow graphs—visual representations showing nodes (statements) and edges (possible transitions)—to identify test scenarios that cover different paths, ensuring comprehensive validation of program logic.