Behaviour-Driven Development (BDD)

Bridging Business and Technical Teams

Mehdi Lotfinejad

🎯 Learning Objectives

By the end of this lesson, you will be able to:

1. **Explain** what Behaviour-Driven Development (BDD) is and how it differs from Test-Driven Development (TDD)
2. **Implement** the "Three Amigos" collaboration approach to define features before development
3. **Write** BDD scenarios using the Given-When-Then format with proper structure
4. **Apply** BDD principles using tools like pytest-bdd to test application behavior
5. **Identify** when BDD adds value and when alternative approaches may be more appropriate

The Communication Gap Problem

Scenario: Building a discount code feature for e-commerce

Different Perspectives:

- **Product Manager:** "Users should apply discount codes"
- **Developer:** Thinks database schemas, API endpoints
- **Tester:** Wonders about expired codes, edge cases

The Problem:

- Each person has different mental model
- Leads to rework and bugs
- Features don't meet business needs

Communication gaps are expensive—discovered late in testing or production.

What is BDD?

"BDD is a way for software teams to work that closes the gap between business people and technical people by creating a shared language for describing system behavior through concrete examples written in plain language." — Cucumber Documentation

Key Characteristics:

- **Collaborative methodology** involving business, development, and testing
- **Concrete examples** rather than abstract requirements
- **Plain language** that everyone can understand
- **Shared understanding** built before coding begins

BDD vs. TDD: The Shift in Focus

Test-Driven Development (TDD):

- Question: "Does this code work correctly?"
- Focus: Code correctness
- Audience: Developers
- Tests: Technical unit tests

Behaviour-Driven Development (BDD):

- Question: "Does the system behave as expected?"
- Focus: Business behavior
- Audience: Everyone
- Scenarios: Business-readable

> BDD evolved from TDD but shifts focus from "testing" to "behavior" and from developers to the whole team.

Why BDD Matters

Benefits of BDD:

- **Reduces rework** → Catches misunderstandings before coding
- **Builds shared ownership** → Everyone contributes to defining features
- **Creates living documentation** → Scenarios stay synchronized with code
- **Improves collaboration** → Business and technical teams speak same language
- **Focuses on value** → Ensures you're building the right thing

BDD prevents the costly "build it wrong, then rebuild it right" cycle.

🤝 The Three Amigos

At the heart of BDD is collaborative discovery called **"The Three Amigos"**:

1. 👔 **Business Perspective** (Product Owner, Business Analyst)
    - Defines what value the feature should deliver
    - Represents user needs and business goals
2. 💻 **Development Perspective** (Developer, Architect)
    - Considers technical constraints and implementation
    - Asks about edge cases and feasibility
3. 🧪 **Testing Perspective** (QA Engineer, Tester)
    - Explores edge cases and quality concerns
    - Thinks about what could go wrong

The Three Amigos in Action

Specification Workshop Process:

1. **Meet before development** starts
2. **Discuss concrete examples** of feature behavior
3. **Ask questions** from all perspectives
4. **Document scenarios** in Given-When-Then format
5. **Build shared understanding** of what to build

Example Discussion:

- **Business:** "We need to handle discount codes"
- **Developer:** "What happens if someone applies two codes?"
- **Tester:** "What if the code expired yesterday?"

> These questions surface requirements that would otherwise be discovered in production!

Concrete Examples Over Abstractions

❌ **Abstract (Unclear):**

- "Users can search for products"
- "The system should be user-friendly"
- "Handle discount codes"

✅ **Concrete (Clear):**

- "When I search for 'red shoes' with 5 in stock, I see all 5 results"
- "When I enter invalid email, I see error within 2 seconds"
- "When I apply 'SAVE20', my $100 cart becomes $80"

Concrete examples are easier to understand, validate, and automate.

📝 Given-When-Then Structure

BDD scenarios follow a specific format that makes behavior explicit:

Given → Sets up initial context or preconditions

- Describes system state before behavior occurs
- Example: "Given a user has 3 items in their cart"

When → Describes the action or event

- What the user does or what happens
- Example: "When the user applies discount code 'SAVE20'"

Then → Specifies expected outcome

- What should happen as a result
- Example: "Then the total price should be reduced by 20%"

Gherkin Language

BDD scenarios are written in **Gherkin**: a business-readable, domain-specific language.

```
Feature: User Authentication
  As a registered user
  I want to log in to my account
  So that I can access my personal dashboard

  Scenario: Successful login with valid credentials
    Given a user exists with email "user@example.com" and password "SecurePass123"
    And the user is on the login page
    When the user enters email "user@example.com"
    And the user enters password "SecurePass123"
    And the user clicks the "Login" button
    Then the user should be redirected to the dashboard
    And the user should see a welcome message "Welcome back!"
```

Why Given-When-Then Works

Three key benefits:

1. **Separates concerns** → Setup (Given) from action (When) from verification (Then)
2. **Business language** → Non-technical stakeholders can read and validate behavior
3. **Automation-ready** → Structured enough for tools to parse and execute

> This format creates a clear narrative that both humans and automation tools understand.

Behavior-Focused vs. Implementation-Focused

❌ **Implementation-Focused:**

```
Given the database has
  user record id=123
When authenticate() method
  is called
Then session token should
  be stored in Redis
```

✅ **Behavior-Focused:**

```
Given I am a
  registered user
When I log in with
  valid credentials
Then I should access
  my account
```

> **Focus on what users experience,** not how the system implements it internally.

Scenario Outline: DRY Testing

Test multiple cases without duplication using **Scenario Outline**:

```
Scenario Outline: Applying discount codes
  Given a shopping cart with total <original_price>
  When the user applies discount code "<code>"
  Then the final price should be <final_price>

  Examples:
    | original_price | code      | final_price |
    | 100.00         | SAVE20    | 80.00       |
    | 100.00         | SAVE50    | 50.00       |
    | 100.00         | INVALID   | 100.00      |
```

Each row becomes a separate test execution with the same steps.

🔄 The BDD Workflow

BDD follows a structured 5-step cycle:

1. **Discovery** → Three Amigos discuss feature, create scenarios
2. **Formulation** → Refine scenarios into Gherkin format
3. **Automation** → Write step definitions connecting scenarios to code
4. **Execution** → Run scenarios as automated tests
5. **Living Documentation** → Scenarios stay synchronized with system

This workflow integrates collaboration, specification, and automation into one process.

Step 1: Discovery

The Three Amigos meet to:

- Explore business goals
- Discuss concrete examples
- Identify edge cases
- Ask questions from all perspectives

Output: Set of concrete scenarios describing feature behavior

> This conversation is more valuable than the scenarios themselves—it builds shared understanding.

Step 2: Formulation

Refine scenarios into structured Gherkin:

```
Feature: Shopping Cart Management
  As a customer
  I want to manage items in my shopping cart
  So that I can purchase products

  Scenario: Adding items to an empty cart
    Given the shopping cart is empty
    When I add a product "Laptop" with price 999.99
    Then the cart should contain 1 item
    And the cart total should be 999.99
```

These become executable specifications that serve as both documentation and tests.

Step 3: Automation

Write step definitions connecting Gherkin to Python code:

```python
from pytest_bdd import scenarios, given, when, then, parsers
from decimal import Decimal

scenarios('../features/shopping_cart.feature')

@given('the shopping cart is empty')
def empty_cart(cart):
    """Set up initial state: ensure cart has no items"""
    cart.clear()
    assert cart.item_count() == 0

@when(parsers.parse('I add a product "{product_name}" with price {price:f}'))
def add_product(cart, product_name, price):
    """Perform the action: add a product to the cart"""
    cart.add_item(product_name, Decimal(str(price)))
```

Step Definitions (continued)

```
@then(parsers.parse('the cart should contain {count:d} item'))
def verify_item_count(cart, count):
    """Verify outcome: check cart has expected number of items"""
    actual_count = cart.item_count()
    assert actual_count == count, \
        f"Expected {count} items, but cart has {actual_count}"

@then(parsers.parse('the cart total should be {expected_total:f}'))
def verify_cart_total(cart, expected_total):
    """Verify outcome: check cart total matches expected value"""
    actual_total = cart.get_total()
    expected = Decimal(str(expected_total))
    assert actual_total == expected, \
        f"Expected total {expected}, but got {actual_total}"
```

Step 4: Execution

Run scenarios as automated tests:

```
$ pytest tests/step_defs/test_shopping_cart.py -v

tests/step_defs/test_shopping_cart.py::test_adding_items_to_an_empty_cart PASSED

Feature: Shopping Cart Management
  Scenario: Adding items to an empty cart
    Given the shopping cart is empty
    When I add a product "Laptop" with price 999.99
    Then the cart should contain 1 item
    And the cart total should be 999.99
```

> When scenarios pass, everyone knows the feature works as expected!

Step 5: Living Documentation

Scenarios remain as living documentation:

- **Always up-to-date** → Scenarios execute against real code
- **Never outdated** → When behavior changes, scenarios are updated
- **Readable by all** → Business stakeholders can validate behavior
- **Single source of truth** → Documentation and tests are the same

Unlike traditional requirements documents that become stale, BDD scenarios stay synchronized with the system.

Pytest Fixtures for Test Isolation

Use fixtures to provide clean state for each scenario:

```python
# tests/conftest.py
import pytest
from shopping_cart import ShoppingCart

@pytest.fixture
def cart():
    """
    Fixture provides a fresh ShoppingCart instance for each scenario
    This ensures test isolation - each scenario starts with clean state,
    preventing one test from affecting another
    """
    return ShoppingCart()
```

Each scenario gets its own cart instance → no state leakage between tests.

📋 Practical Example: User Registration

Let's walk through a complete BDD example for user registration.

Feature File:

```
Feature: User Registration with Email Verification
  As a new visitor
  I want to register for an account
  So that I can access member features

  Scenario: Successful registration with valid email
    Given I am on the registration page
    When I enter email "newuser@example.com"
    And I enter password "StrongPass123!"
    And I enter password confirmation "StrongPass123!"
    And I click the "Register" button
    Then I should see a message "Please check your email to verify your account"
    And a verification email should be sent to "newuser@example.com"
```

User Registration: Error Scenario

```
Scenario: Registration fails with mismatched passwords
  Given I am on the registration page
  When I enter email "user@example.com"
  And I enter password "Password123!"
  And I enter password confirmation "DifferentPass456!"
  And I click the "Register" button
  Then I should see an error "Passwords do not match"
  And no verification email should be sent
```

> Both success and failure paths are documented in business-readable scenarios.

Step Definitions: Page Navigation

```python
from pytest_bdd import scenarios, given, when, then, parsers

scenarios('../features/user_registration.feature')

@given('I am on the registration page')
def on_registration_page(browser, base_url):
    """
    Navigate to registration page
    We verify the page loaded correctly by checking the title,
    which catches navigation failures early
    """
    browser.get(f"{base_url}/register")
    assert "Register" in browser.title
```

Step Definitions: User Input

```
@when(parsers.parse('I enter email "{email}"'))
def enter_email(browser, email, registration_data):
    """
    Fill in the email field
    We store email in registration_data dict so later steps
    can verify the correct email was used
    """
    email_field = browser.find_element_by_id("email")
    email_field.clear()  # Clear existing value for clean state
    email_field.send_keys(email)
    registration_data['email'] = email

@when(parsers.parse('I enter password "{password}"'))
def enter_password(browser, password):
    """Fill in password field"""
    password_field = browser.find_element_by_id("password")
    password_field.clear()
    password_field.send_keys(password)
```

Step Definitions: Verification

```
@then(parsers.parse('I should see a message "{expected_message}"'))
def verify_success_message(browser, expected_message):
    """
    Verify success message appears on page
    We wait up to 5 seconds because the server may need time
    to process the registration asynchronously
    """
    from selenium.webdriver.support.ui import WebDriverWait
    from selenium.webdriver.support import expected_conditions as EC
    from selenium.webdriver.common.by import By

    message_element = WebDriverWait(browser, 5).until(
        EC.presence_of_element_located((By.CLASS_NAME, "success-message"))
    )
    assert expected_message in message_element.text
```

Step Definitions: Email Verification

```python
@then(parsers.parse('a verification email should be sent to "{email}"'))
def verify_email_sent(email_service, email):
    """
    Verify verification email was sent
    We check both recipient and subject to ensure
    the correct type of email was sent
    """
    sent_emails = email_service.get_sent_emails()
    matching_emails = [
        e for e in sent_emails
        if e['to'] == email and 'verify' in e['subject'].lower()
    ]
    assert len(matching_emails) == 1, \
        f"Expected 1 verification email to {email}, found {len(matching_emails)}"
```

Complete Workflow Demonstration

How it all connects:

1. **Business stakeholders** read feature file and validate behavior
2. **Developers** implement step definitions that exercise actual code
3. **Testers** run scenarios to verify system behavior
4. **Everyone** knows feature works when scenarios pass

This creates a tight feedback loop where business requirements drive technical implementation.

🚫 Common Pitfall 1: Implementation-Focused Scenarios

Problem: Writing scenarios that describe internal implementation.

Bad Example:

```
Given the database has a user record with id=123
When the authenticate() method is called with credentials
Then the session token should be stored in Redis
```

Good Example:

```
Given I am a registered user
When I log in with valid credentials
Then I should access my account
```

Fixing Implementation Focus

Ask the right question:

- ❌ "What does the code do?"
- ✅ "What does the user see or experience?"

Focus on:

- User-observable behavior
- Business outcomes
- External perspective

> **Remember:** Scenarios should survive implementation changes—if you switch from Redis to Memcached, scenarios shouldn't change.

🚫 Common Pitfall 2: Overly Complex Scenarios

Problem: Scenarios with dozens of steps testing multiple behaviors.

Bad Example:

```
Scenario: Complete user journey
  Given I register an account
  And I verify my email
  And I log in
  And I add 5 products to cart
  And I apply a discount
  And I enter shipping address
  And I enter payment details
  And I complete purchase
  Then... (10 more assertions)
```

Fixing Complex Scenarios

Keep scenarios focused:

- One scenario → One behavior
- Maximum 5-7 steps as guideline
- Split complex scenarios into multiple focused ones

Use Scenario Outline for variations:

```
Scenario Outline: Login validation
  Given I am on the login page
  When I enter email "<email>"
  Then I should see "<message>"

  Examples:
    | email            | message              |
    | valid@email.com | Welcome              |
    | invalid          | Invalid email format |
```

🚫 Common Pitfall 3: Skipping Three Amigos

Problem: Developers write scenarios alone without collaboration.

Consequences:

- Business perspective missing
- Edge cases not discussed
- Scenarios don't reflect actual needs
- Defeats primary purpose of BDD

Solution:

- Schedule regular Three Amigos sessions (30 min minimum)
- Use video calls for distributed teams
- Make it part of your definition of ready

> Even 30 minutes of discussion prevents days of rework!

🚫 Common Pitfall 4: Using BDD for Everything

Problem: Applying BDD to unit tests and technical APIs.

When BDD adds overhead:

- Low-level unit tests
- Technical API contracts
- Internal utility functions
- Performance testing

When BDD adds value:

- User-facing features
- Acceptance tests
- Complex business rules
- Features requiring stakeholder validation

Choosing the Right Testing Approach

Use BDD for:

- Shopping cart checkout
- User registration
- Payment processing
- Search functionality

Use Unit Tests for:

- String parsing function
- Math calculations
- Data transformations
- Algorithm implementation

> **BDD is not a replacement for all testing**—it complements unit and integration tests.

🚫 Common Pitfall 5: Scenarios as Afterthought

Problem: Write code first, then create scenarios to document what was built.

Why this fails:

- Reverses the BDD workflow
- Scenarios don't guide development
- Loses collaboration benefit
- Just becomes documentation

Correct workflow:

1. Write scenarios BEFORE implementation
2. Use scenarios to guide what you build
3. Scenarios drive development, not just verify it

🚫 Common Pitfall 6: Ignoring Maintenance

Problem: Scenarios become outdated or accumulate duplicates.

How to fix:

- **Treat scenarios as first-class code** → Review and refactor regularly
- **Update immediately** when behavior changes
- **Remove duplicates** → Consolidate similar scenarios
- **Archive obsolete scenarios** → Don't let dead scenarios accumulate

> Schedule quarterly scenario reviews to keep them clean and relevant.

📊 BDD vs. Traditional Requirements

Traditional Requirements:

- Prose descriptions
- "The system shall..."
- Often become outdated
- Not executable
- Separate from tests

BDD Scenarios:

- Concrete examples
- "Given-When-Then"
- Stay synchronized
- Executable tests
- Documentation = Tests

BDD vs. TDD

Aspect	BDD	TDD
Question	"Does system behave correctly?"	"Does this code work?"
Focus	User behavior	Code correctness
Language	Business terms	Technical terms
Audience	Everyone	Developers
Level	Acceptance tests	Unit tests
Collaboration	High	Low

> BDD and TDD are complementary—use both for different purposes!

📊 BDD vs. Manual Testing

Manual Testing:

- Human testers follow scripts
- Slow and expensive
- Catches UI/UX issues
- Good for exploratory testing
- Not repeatable

BDD:

- Automated execution
- Fast and consistent
- Business-readable
- Good for regression testing
- Highly repeatable

Comparison Summary Table

Aspect	BDD	TDD	Manual Testing
Focus	User behavior	Code correctness	User experience
Language	Business terms	Technical terms	Natural language
Automation	Yes	Yes	No
Collaboration	High	Low	Medium
Best for	Acceptance tests	Unit tests	Exploratory testing

When to Use BDD

BDD works best when:

- Features have clear business value
- Collaboration between roles adds value
- Stakeholders need to validate behavior
- Living documentation is valuable
- Requirements are complex

Consider alternatives when:

- Testing purely technical components
- No business stakeholders involved
- Simple features with obvious behavior
- Unit-level testing is sufficient

🎯 Key Takeaways

- **BDD bridges gaps** between business and technical teams through shared language
- **Three Amigos approach** ensures all perspectives (business, developer, tester) are considered early
- **Given-When-Then format** written in Gherkin creates clear, executable scenarios
- **Focus on behavior,** not implementation—scenarios should survive code changes
- **Living documentation** stays synchronized through automated execution

🎯 Key Takeaways (continued)

- **Tools like pytest-bdd** connect business-readable scenarios to executable code
- **BDD is most valuable** for acceptance testing of user-facing features
- **Collaborative conversation** is more important than the scenarios themselves
- **Write scenarios before implementation** to guide development
- **Not everything needs BDD**—use it where it adds value

📝 Practice Quiz: Question 1

What is the primary purpose of BDD?

A) To write more unit tests

B) To replace manual testing completely

C) To close the gap between business and technical teams

D) To automate all testing activities

📝 Answer 1

Correct: C) To close the gap between business and technical teams

Explanation:

BDD's primary purpose is to close the gap between business people and technical people by creating a shared language for describing system behavior.

> While BDD does involve automation, its core value is in improving collaboration and ensuring everyone has the same understanding of what the system should do.

📝 Practice Quiz: Question 2

In the Given-When-Then format, what does the "Given" step represent?

A) The expected outcome
B) The action the user performs
C) The initial context or preconditions
D) The error handling logic

📝 Answer 2

Correct: C) The initial context or preconditions

Explanation:

The "Given" step sets up the initial context or preconditions before the behavior occurs. It describes the state of the system before any action is taken.

Examples:

- "Given a user is logged in"
- "Given the shopping cart contains 3 items"
- "Given a user has an active subscription"

📝 Practice Quiz: Question 3

Who are the "Three Amigos" in BDD?

A) Three developers who pair program

B) Business person, developer, and tester

C) Product manager, designer, and engineer

D) Three QA engineers who review tests

📝 Answer 3

Correct: B) Business person, developer, and tester

Explanation:

The Three Amigos are:

- **Business person** (or product owner)
- **Developer**
- **Tester**

This combination ensures all three perspectives—business value, technical feasibility, and quality concerns—are considered before development starts.

These three roles collaborate to define features before any code is written.

📝 Practice Quiz: Question 4

When is BDD LEAST valuable?

A) When testing user-facing features
B) When testing low-level API contracts
C) When business stakeholders need to validate behavior
D) When creating living documentation

📝 Answer 4

Correct: B) When testing low-level API contracts

Explanation:

BDD has limited value for testing low-level API contracts because these are best expressed in technical terms like JSON schemas rather than business-readable scenarios.

> APIs are technical interfaces where the abstraction of Given-When-Then adds overhead without improving clarity.

BDD is most valuable for user-facing features with clear business value.

📝 Practice Quiz: Question 5

What is the main difference between BDD and TDD?

A) BDD uses Python while TDD uses Java

B) BDD focuses on behavior while TDD focuses on code correctness

C) BDD is automated while TDD is manual

D) BDD is only for web applications

📝 Answer 5

Correct: B) BDD focuses on behavior while TDD focuses on code correctness

Explanation:

- **BDD** asks: "Does the system behave as the business expects?"
- **TDD** asks: "Does this code work correctly?"

Key points:

- Both use automation
- Both can be used with any programming language
- BDD can be applied to any type of application
- They complement each other—use both!

🔧 BDD Tools Ecosystem

Popular BDD Frameworks:

Language	Framework
Python	pytest-bdd, behave
JavaScript	Cucumber.js, Jest-Cucumber
Java	Cucumber-JVM, JBehave
Ruby	Cucumber, RSpec
C#	SpecFlow, BDDfy

All support Gherkin syntax and provide step definition mechanisms.

Installing pytest-bdd

Setup for Python projects:

```
# Install pytest-bdd
pip install pytest-bdd

# Project structure
project/
├── features/
│   └── shopping_cart.feature
├── tests/
│   ├── conftest.py
│   └── step_defs/
│       └── test_shopping_cart.py
└── shopping_cart.py
```

Running BDD Tests

Execute scenarios with pytest:

```
# Run all BDD tests
pytest tests/step_defs/ -v

# Run specific feature
pytest tests/step_defs/test_shopping_cart.py -v

# Run with coverage
pytest tests/step_defs/ --cov=shopping_cart

# Generate HTML report
pytest tests/step_defs/ --html=report.html
```

Best Practices for Writing Scenarios

DO:

- ✅ Focus on user-observable behavior
- ✅ Use business language, not technical jargon
- ✅ Keep scenarios short and focused (5-7 steps)
- ✅ Use Scenario Outline for variations
- ✅ Write scenarios before implementation

DON'T:

- ❌ Describe implementation details
- ❌ Create overly complex scenarios
- ❌ Skip the Three Amigos conversation
- ❌ Let scenarios become outdated
- ❌ Use BDD for everything

BDD Workflow Summary

```
1. Discovery
   ↓ (Three Amigos discuss)
2. Formulation
   ↓ (Write Gherkin scenarios)
3. Automation
   ↓ (Implement step definitions)
4. Execution
   ↓ (Run as automated tests)
5. Living Documentation
   ↓ (Scenarios stay synchronized)

→ Feature Complete! ✓
```

Real-World BDD Success Stories

When BDD adds significant value:

- **E-commerce checkout flows** → Complex business rules, multiple stakeholders
- **Financial calculations** → Precise requirements, regulatory compliance
- **User authentication** → Security concerns, multiple scenarios
- **Booking systems** → Complex workflows, business validation needed
- **Payment processing** → Critical functionality, clear acceptance criteria

> BDD excels where business rules are complex and stakeholder validation is critical.

Getting Started with BDD

Steps to adopt BDD in your team:

1. **Start small** → Pick one feature to try BDD
2. **Schedule Three Amigos** → Get all perspectives involved
3. **Write scenarios together** → Build shared understanding
4. **Automate incrementally** → Don't wait for perfect automation
5. **Review and refine** → Learn from each iteration
6. **Expand gradually** → Add BDD to more features over time

> Don't try to convert all existing tests to BDD—focus on new features first.

Common Questions About BDD

Q: Does BDD replace unit tests?

A: No—BDD complements unit tests. Use unit tests for code logic, BDD for user behavior.

Q: How long do Three Amigos sessions take?

A: 30 minutes to 2 hours per feature, depending on complexity.

Q: Who writes step definitions?

A: Developers, but scenarios are written collaboratively.

Q: Can we use BDD for APIs?

A: Possible, but often adds overhead without clear benefit for technical APIs.

📚 Additional Resources

Official Documentation:

- Cucumber BDD: https://cucumber.io/docs/bdd/
- pytest-bdd: https://pytest-bdd.readthedocs.io/
- Behave (Python): https://behave.readthedocs.io/

Learning Materials:

- "BDD in Action" by John Ferguson Smart
- "The Cucumber Book" by Matt Wynne and Aslak Hellesøy
- Cucumber School: https://school.cucumber.io/

Community:

- Cucumber Community: https://cucumber.io/community
- BDD discussions on Stack Overflow

Quick Reference: Gherkin Keywords

Keyword	Purpose	Example
Feature	Describes feature	`Feature: User Login`
Scenario	Single test case	`Scenario: Valid login`
Given	Preconditions	`Given I am logged in`
When	Action	`When I click logout`
Then	Expected outcome	`Then I see login page`
And	Additional steps	`And I see welcome message`
But	Negative condition	`But I don't see errors`

Quick Reference: Step Definition Patterns

```
# Simple step
@given('the cart is empty')
def empty_cart(cart):
    cart.clear()

# Step with string parameter
@when(parsers.parse('I add "{item}"'))
def add_item(cart, item):
    cart.add(item)

# Step with typed parameter
@then(parsers.parse('the total is {amount:f}'))
def verify_total(cart, amount):
    assert cart.total() == amount
```

Final Thoughts

BDD is about collaboration first, automation second.

What you've learned:

- How BDD bridges business and technical teams
- The Three Amigos collaborative approach
- Writing effective Given-When-Then scenarios
- Implementing BDD with pytest-bdd
- When BDD adds value and when to use alternatives

Next steps:

- Try BDD on your next feature
- Schedule a Three Amigos session
- Write scenarios before coding

🎉 Lesson Complete!

You now understand:

- ✅ What BDD is and how it differs from TDD
- ✅ The Three Amigos collaboration approach
- ✅ How to write Given-When-Then scenarios in Gherkin
- ✅ How to implement BDD with pytest-bdd
- ✅ When to use BDD and when alternatives are better
- ✅ Common pitfalls and how to avoid them

> **Start practicing!** The best way to learn BDD is to try it on a real feature with your team. Begin with a simple scenario and build from there.

Collaboration creates better software!