



Behavior Driven Development

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

Behavior Driven Development (BDD) is a software development process that originally emerged from Test Driven Development (TDD). BDD uses examples to illustrate the behavior of the system that are written in a readable and understandable language for everyone involved in the development.

Audience

BDD focuses on providing a shared process and shared tools promoting communication among software developers and business analysts to collaborate on software development, with the aim of delivering products with business value. Hence, this tutorial is going to be useful for software developers as well as business analysts at every level.

Prerequisites

Before you start proceeding with this tutorial, we are assuming that you are already aware of the basics of testing and have some hands-on experience of some testing tools. If you are not well aware of these concepts, then we will suggest you to go through our short tutorial Software Testing.

Copyright & Disclaimer

© Copyright 2016 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial	i
Audience.....	i
Prerequisites.....	i
Copyright & Disclaimer	i
Table of Contents	ii
1. BDD – Introduction	1
BDD – Key features	1
Origin of BDD	1
BDD Practices	2
Agile BDD	2
Agile Manifesto and BDD.....	3
2. BDD – Test Driven Development	4
Why Testing?	4
Challenges with Test-Last Approach.....	5
Test-First Approach	5
Red-Green-Refactor Cycle	6
TDD Process Steps	7
Advantages of TDD	7
Disadvantages of TDD.....	8
Misconceptions about TDD	8
Acceptance TDD	9
TDD Vs BDD	10
3. BDD – TDD in a BDD Way	11
Story and Scenarios	12
Development Cycle.....	12
4. BDD – Specifications by Example	14
Specification by Example – Overview	14
Use of SbE.....	14
Advantages of SbE	15
Applications of SbE	15
SbE and Acceptance Testing.....	15
SbE – A Set of Process Patterns	16
Collaborative Specification	17
Illustrating Specification using Examples.....	17
Refining the Specification	18
Automating Examples.....	18
Validating Frequently	18
Living Documentation.....	19
Anti-patterns	19
Solution to the Problems - Quality	20
Tools	21
5. BDD – Tools.....	22
Cucumber	22
SpecFlow.....	23

Behave	23
Lettuce.....	23
Concordion	24
6. BDD – BDD and Cucumber	25
Typical Cucumber Acceptance Test	26
Working of Cucumber	26
Mapping Steps and Step Definitions	27
7. BDD – Gherkin	30
Gherkin Format and Syntax	30
8. BDD – BDD and SpecFlow.....	34
Feature Elements and SpecFlow	34
Multiple Scenario Steps	34
Tags.....	35
Scenario Outlines.....	35

1. BDD – Introduction

Behavior Driven Development (BDD) is a software development process that originally emerged from Test Driven Development (TDD).

According to Dan North, who is responsible for the evolution of BDD, “BDD is using examples at multiple levels to create a shared understanding and surface uncertainty to deliver software that matter.”

BDD uses examples to illustrate the behavior of the system that are written in a readable and understandable language for everyone involved in the development. These examples include –

- Converted into executable specifications.
- Used as the acceptance tests.

BDD – Key features

Behavior Driven Development focuses on –

- Providing a shared process and shared tools promoting communication to the software developers, business analysts and stakeholders to collaborate on software development, with the aim of delivering product with business value.
- What a system should do and not on how it should be implemented.
- Providing better readability and visibility.
- Verifying not only the working of the software but also that it meets the customer’s expectations.

Origin of BDD

The cost to fix a defect increases multifold if the defect is not detected at the right time and fixed as and when it is detected. Consider the following example.

Cost To Fix		Time detected				
		Requirements	Design	Building	Testing	Post-Release
Time Introduced	Requirements	1x	3x	5-10x	10x	10-100x
	Design	–	1x	10x	15x	25-100x
	Building	–	–	1x	10x	10-25x

This shows that unless requirements are obtained correctly, it would be expensive to fix the defects resulting from misunderstanding the requirements at a later stage. Further, the end product may not meet the customer’s expectations.

The need of the hour is a development approach that –

- Is based on the requirements.
- Focuses on requirements throughout the development.
- Ensures that the requirements are met.

A development approach that can take care of the above-mentioned requirements is BDD. Thus, Behavior Driven Development –

- Derives examples of different expected behaviors of the system.
- Enables writing the examples in a language using the business domain terms to ensure easy understanding by everyone involved in the development including the customers.
- Gets the examples ratified with customer from time to time by means of conversations.
- Focuses on the customer requirements (examples) throughout the development.
- Uses examples as acceptance tests.

BDD Practices

The two main practices of BDD are –

- Specification by Example (SbE)
- Test Driven Development (TDD)

Specification by Example

Specification by Example (SbE) uses examples in conversations to illustrate the business rules and the behavior of the software to be built.

Specification by Example enables the product owners, business analysts, testers and the developers to eliminate common misunderstandings about the business requirements.

Test Driven Development

Test Driven Development, in the context of BDD, turns examples into human readable, executable specifications.

The developers use these specifications as a guide to implement increments of new functionality. This, results in a lean codebase and a suite of automated regression tests that keep the maintenance costs low throughout the lifetime of the software.

Agile BDD

In Agile software development, BDD method is used to come to a common understanding on the pending specifications.

The following steps are executed in Agile BDD –

- The developers and the product owner collaboratively write pending specifications in a plain text editor.
- The product owner specifies the behaviors they expect from the system.
- The developers –
 - Fill the specifications with these behavior details.
 - Ask questions based on their understanding of the system.
- The current system behaviors are considered to see if the new feature will break any of the existing features.

Agile Manifesto and BDD

The Agile Manifesto states the following –

We are uncovering better ways of developing software by doing it and helping others do it. Through this work, we have come to value –

- **Individuals and interactions** over Processes and tools
- **Working software** over Comprehensive documentation
- **Customer collaboration** over Contract negotiation
- **Responding to change** over Following a plan

That is, while there is value in the items on the right, we value the items on the left more.

BDD aligns itself to the Agile manifesto as follows –

Agile Manifesto	BDD Alignment
Individuals and interactions over processes and tools.	BDD is about having conversations.
Working software over comprehensive documentation.	BDD focuses on making it easy to create software that is of business value.
Customer collaboration over contract negotiation.	BDD focuses on scenarios based on ideas with continuous communication with the customer as the development progresses. It is not based on any promises.
Responding to change over following a plan.	BDD focuses on continuous communication and collaboration that facilitates absorption of changes.

2. BDD – Test Driven Development

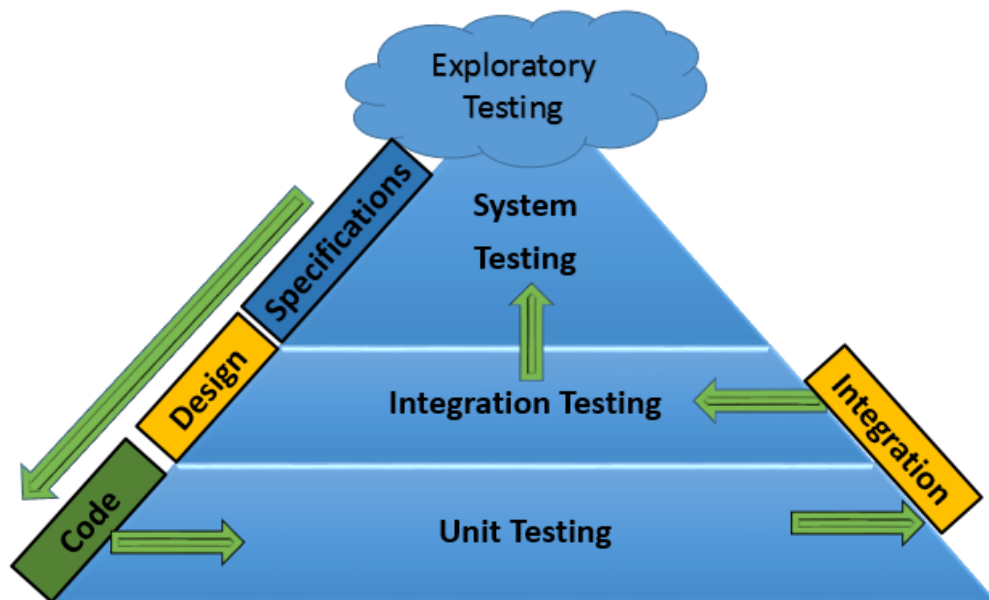
When you look at any reference on Behavior Driven Development, you will find the usage of phrases such as “BDD is derived from TDD”, “BDD and TDD”. To know how BDD came into existence, why it is said to be derived from TDD and what is BDD and TDD, you have to have an understanding of TDD.

Why Testing?

To start, let us get into the fundamentals of testing. The purpose of testing is to ensure that the system that is built is working as expected. Consider the following example.

Cost To Fix		Time detected				
		Requirements	Design	Building	Testing	Post-Release
Time Introduced	Requirements	1x	3x	5-10x	10x	10-100x
	Design	–	1x	10x	15x	25-100x
	Building	–	–	1x	10x	10-25x

Hence, by experience we have learnt that uncovering a defect as and when it is introduced and fixing it immediately would be cost effective. Therefore, there is a necessity of writing test cases at every stage of development and testing. This is what our traditional testing practices have taught us, which is often termed as Test-early.



This testing approach is termed as the Test-Last approach as testing is done after the completion of a stage.

Challenges with Test-Last Approach

The Test-Last approach was followed for quite some time in the software development projects. However, in reality, with this approach, as testing has to wait till the particular stage is completed, often it is overlooked because of-

- The delays in the completion of the stage.
- Tight time schedules.
- Focus on delivery on time, skipping testing.

Further, in the Test-Last approach, Unit testing, that is supposed to be done by the developers is often skipped. The various reasons found are based on the mind-set of the developers –

- They are developers and not testers.
- Testing is the responsibility of the testers.
- They are efficient in coding and their code would not have defects.

This results in –

- Compromising on the quality of the product delivered.
- Having the accountability for quality on testers only.
- High-costs in fixing the defects, post delivery.
- Inability to obtain customer satisfaction, which would also mean loss of repeat business, thus effecting credibility.

These factors called for a shift in paradigm, to focus on testing. The result was the Test-First approach.

Test-First Approach

The Test-First approach replaces the inside-out (write code and then test) to outside-in (write test and then code) way of development.

This approach is incorporated into the following software development methodologies (that are Agile also)-

- **eXtreme Programming (XP).**
- **Test Driven Development (TDD).**

In these methodologies, the developer designs and writes the Unit tests for a code module before writing a single line of the code module. The developer then creates the code module with the goal of passing the Unit test. Thus, these methodologies use Unit testing to drive the development.

The fundamental point to note that the goal is development based on testing.

Red-Green-Refactor Cycle

Test Driven Development is used to develop the code guided by Unit tests.

Step 1: Consider a code module that is to be written.

Step 2: Write a test.

Step 3: Run the test.

The test fails, as the code is still not written. Hence, Step 2 is usually referred to as write a test to fail.

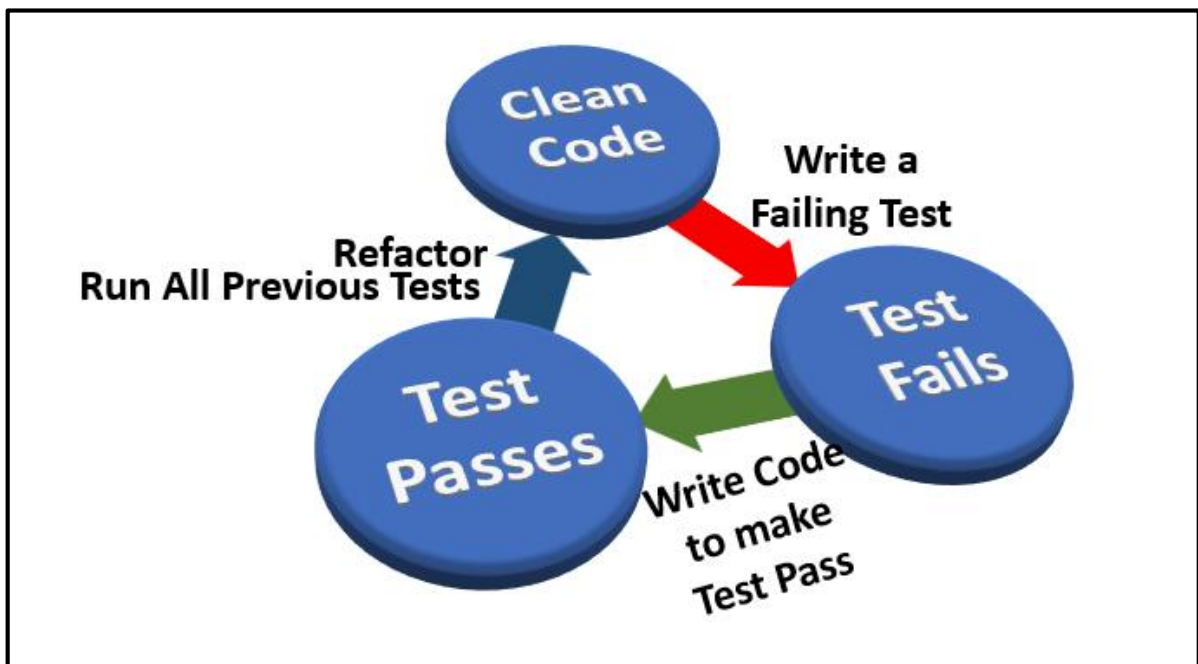
Step 4: Write minimum code possible to pass the test.

Step 5: Run all the tests to ensure that they all still pass. Unit tests are automated to facilitate this step.

Step 6: Refactor.

Step 7: Repeat Step 1 to Step 6 for the next code module.

Each cycle should be very short, and a typical hour should contain many cycles.

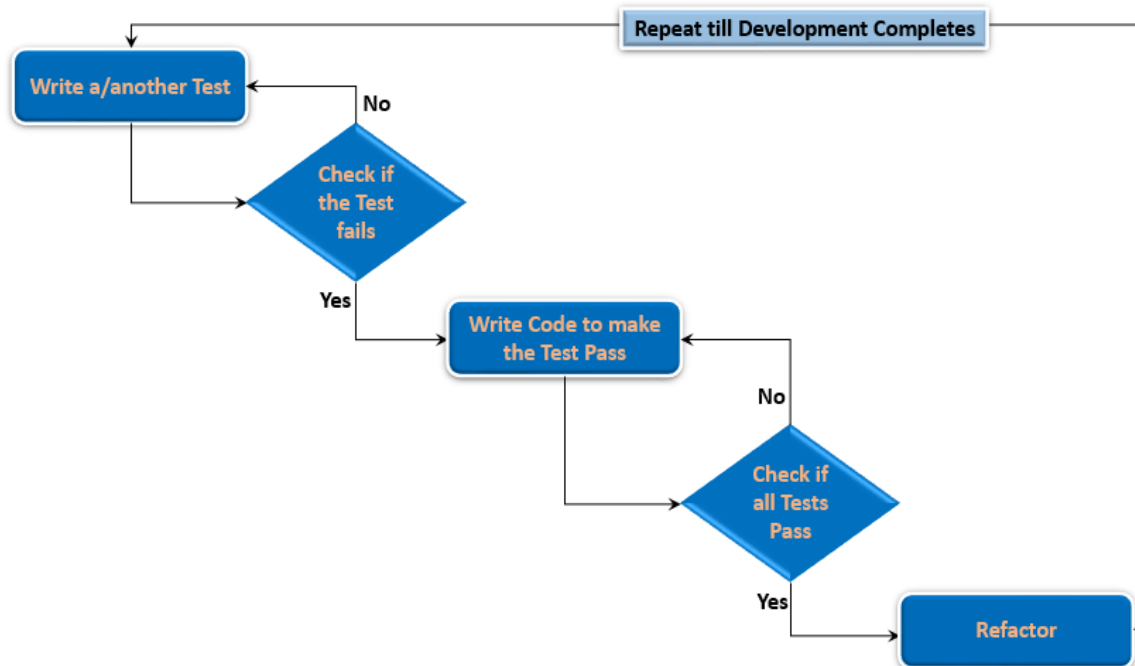


This is also popularly known as the **Red-Green-Refactor** cycle, where –

- **Red:** Writing a test that fails.
- **Green:** Writing code to pass the test.
- **Refactor:** Remove duplication and improve the code to the acceptable standards.

TDD Process Steps

The steps of a TDD process are illustrated below.



Advantages of TDD

The benefits or advantages of Test Driven Development are –

- The developer needs to understand first, what the desired result should be and how to test it before creating the code.
- The code for a component is finished only when the test passes and the code is refactored. This ensures testing and refactoring before the developer moves on to the next test.
- As the suite of Unit tests is run after each refactoring, feedback that each component is still working is constant.
- The Unit tests act as living documentation that is always up to the data.
- If a defect is found, the developer creates a test to reveal that defect and then modify the code so that the test passes and the defect is fixed. This reduces the debugging time. All the other tests are also run and when they pass, it ensures that the existing functionality is not broken
- The developer can make design decisions and refactor at any time and the running of the tests ensures that the system is still working. This makes the software maintainable.
- The developer has the confidence to make any change since if the change impacts any existing functionality, the same is revealed by running the tests and the defects can be fixed immediately.

- On each successive test run, all the previous defect fixes are also verified and the repetition of same defect is reduced.
- As most of the testing is done during the development itself, the testing before delivery is shortened.

Disadvantages of TDD

The starting point is User Stories, describing the behavior of the system. Hence, the developers often face the following questions –

- When to test?
- What to test?
- How to know if a specification is met?
- Does the code deliver business value?

Misconceptions about TDD

The following misconceptions exist in the industry and need clarifications.

Misconception	Clarification
TDD is all about testing and test automation.	TDD is a development methodology using Test-First approach.
TDD does not involve any design.	TDD includes critical analysis and design based on the requirements. The design emerges during development.
TDD is only at Unit level.	TDD can be used at the integration and system levels.
TDD cannot be used by traditional testing projects.	TDD became popular with Extreme Programming and is being used in other Agile methodologies. However, it can be used in traditional testing projects as well.
TDD is a tool.	TDD is a development methodology, and after every new Unit Test passes, it is added to the Automation Test Suite as all the tests need to be run whenever a new code is added or existing code is modified and also after every refactoring. Thus, Test Automation Tools supporting TDD facilitate this process.
TDD means handing Acceptance tests to the developers.	TDD does not mean handing Acceptance Tests to the developers.

Acceptance TDD

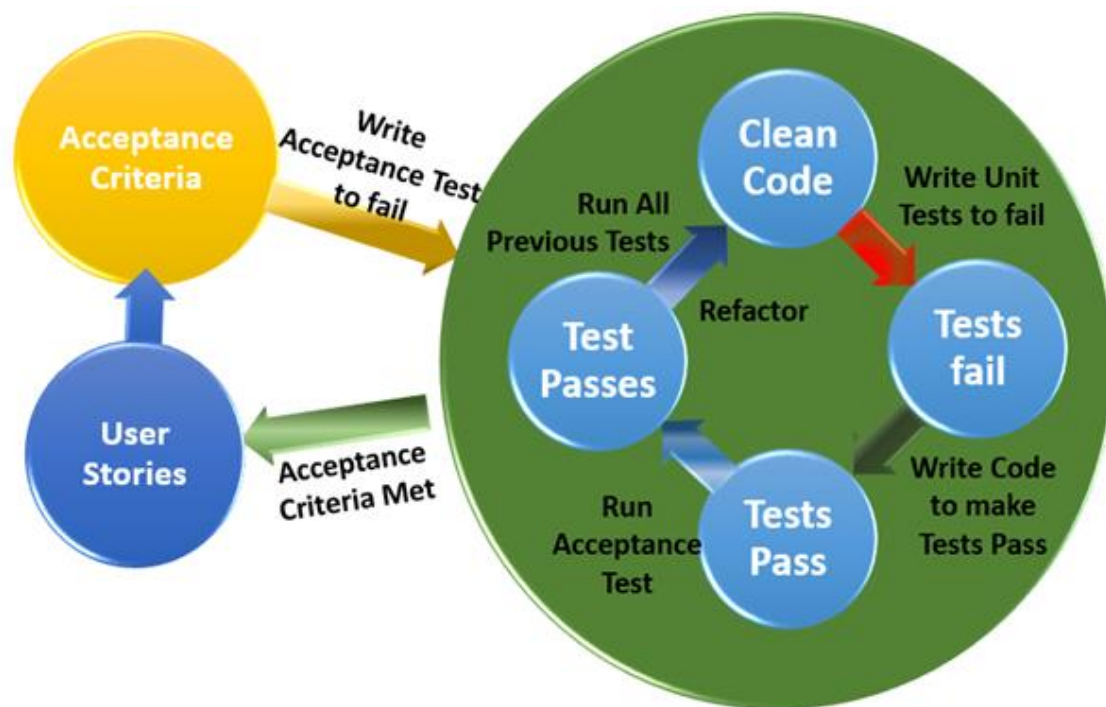
Acceptance Test Driven Development (ATDD) defines Acceptance Criteria and Acceptance Tests during the creation of User Stories, early in development. ATDD focuses on the communication and common understanding among the customers, developers and the testers.

The Key practices in ATDD are as follows –

- Discuss real-world scenarios to build a shared understanding of the domain.
- Use those scenarios to arrive at acceptance criteria.
- Automate Acceptance tests.
- Focus the development on those tests.
- Use the tests as a live specification to facilitate change.

The benefits of using ATDD are as follows –

- Requirements are unambiguous and without functional gaps.
- Others understand the special cases that the developers foresee.
- The Acceptance tests guide the development.



TDD Vs BDD

According to Dan North, programmers normally face the following problems while performing Test Driven Development-

- Where to start
- What to test and what *not* to test
- How much to test in one go
- What to call their tests
- How to understand why a test fails

The solution to all these problems is Behavior Driven Development. It has evolved out of the established agile practices and is designed to make them more accessible and effective for teams, new to agile software delivery. Over time, BDD has grown to encompass the wider picture of agile analysis and automated acceptance testing.

The main **difference between TDD and BDD** is that –

- TDD describes how the software works.
- On the other hand, BDD –
 - Describes how the end user uses the software.
 - Fosters collaboration and communication.
 - Emphasizes on examples of behavior of the System.
 - Aims at the executable specifications derived from the examples

3. BDD – TDD in a BDD Way

In TDD, the term “Acceptance Tests” is misleading. Acceptance tests actually represent the expected behavior of the system. In Agile practices, collaboration of the whole team and interactions with the customer and other stakeholders is emphasized. This has given rise to the necessity of usage of terms that are easily understood by everyone involved in the project.

TDD makes you think about the required behavior and hence the term '**Behavior**' is more useful than the term '**Test**'. BDD is Test Driven Development with a vocabulary that focuses on behavior and not tests.

In the words of Dan North, “I found the shift from thinking in tests to thinking in behavior so profound that I started to refer to TDD as BDD, or Behavior Driven Development.” TDD focuses on how something will work, BDD focuses on why we build it at all.

BDD answers the following questions often faced by the developers-

Question	Answer
Where to start?	outside-in
What to test?	User Stories
What not to test?	anything else

These answers result in the story framework as follows-

Story Framework

As a **[Role]**
I want **[Feature]**
so that **[Benefit]**

This means, 'When a **Feature** is executed, the resulting **Benefit** is to the Person playing the **Role**.'

BDD further answers the following questions-

Question	Answer
How much to test in one go?	very little-focused
What to call their tests?	sentence template
How to understand why a test fails	documentation

These answers result in the Example framework as follows-

Example Framework

Given some initial context,
When an event occurs,
Then ensure some outcomes.

This means, 'Starting with the initial context, when a particular event happens, we know what the outcomes **should be**.'

Thus, the example shows the expected behavior of the system. The examples are used to illustrate different scenarios of the system.

Story and Scenarios

Let us consider the following illustration by Dan North about an ATM system.

Story

As a customer,
I want to withdraw cash from an ATM,
so that I do not have to wait in line at the bank.

Scenarios

There are two possible scenarios for this story.

Scenario 1: Account is in credit

Given the account is in credit
And the card is valid
And the dispenser contains cash
When the customer requests cash
Then ensure the account is debited
And ensure cash is dispensed
And ensure the card is returned

Scenario 2: Account is overdrawn past the overdraft limit

Given the account is overdrawn
And the card is valid
When the customer requests cash
Then ensure a rejection message is displayed
And ensure cash is not dispensed
And ensure the card is returned

The event is same in both the scenarios, but the context is different. Hence, the outcomes are different.

Development Cycle

The Development Cycle for BDD is an **outside-in** approach.

Step 1: Write a high-level (outside) business value example (using Cucumber or RSpec/Capybara) that goes red. (RSpec produces a BDD framework in the Ruby language)

Step 2: Write a lower-level (inside) RSpec example for the first step of implementation that goes red.

Step 3: Implement the minimum code to pass that lower-level example, see it go green.

Step 4: Write the next lower-level RSpec example pushing towards passing Step 1 that goes red.

Step 5: Repeat steps Step 3 and Step 4 until the high-level example in Step 1 goes green.

Note: The following points should be kept in mind-

- **Red/green** state is a permission status.
- When your low-level tests are green, you have the permission to write new examples or refactor existing implementation. You must not, in the context of refactoring, add new functionality/flexibility.
- When your low-level tests are red, you have permission to write or change implementation code only for making the existing tests go green. You must resist the urge to write the code to pass your next test, which does not exist, or implement features you may think are good (customer would not have asked).

4. BDD – Specifications by Example

According to Gojko Adzic, the author of 'Specification by Example', Specification by Example is a set of process patterns that facilitate change in software products to ensure that the right product is delivered efficiently."

Specification by Example is a collaborative approach to define the requirements and business-oriented functional tests for software products based on capturing and illustrating requirements using realistic examples instead of abstract statements.

Specification by Example – Overview

The objective of Specification by Example is to focus on development and delivery of prioritized, verifiable, business requirements. While the concept of Specification by Example in itself is relatively new, it is simply a rephrasing of existing practices.

It supports a very specific, concise vocabulary known as **ubiquitous language** that-

- Enables executable requirements.
- Is used by everyone in the team.
- Is created by a cross-functional team.
- Captures everyone's understanding.

Specification by Example can be used as a direct input into building Automated tests that reflect the business domain. Thus, the focus of Specification by Example is on building the right product and building the product right.

Purpose of Specification by Example

The primary aim of Specification by Example is to build the right product. It focuses on shared understanding, thus establishing a single source of truth. It enables automation of acceptance criteria so that focus is on defect prevention rather than defect detection. It also promotes test early to find the defects early.

Use of SbE

Specification by Example is used to illustrate the expected system behavior that describes business value. The illustration is by means of concrete and real life examples. These examples are used to create executable requirements that are –

- Testable without translation.
- Captured in live documentation.

Following are the reasons why we use examples to describe particular specifications –

- They are easier to understand.
- They are harder to misinterpret.

Advantages of SbE

The advantages of using Specification by Example are –

- Increased quality
- Reduced waste
- Reduced risk of production defects
- Focused effort
- Changes can be made more safely
- Improved business involvement

Applications of SbE

Specification by Example find applications in –

- Either complex business or complex organization.
- Does not work well for purely technical problems.
- Does not work well for UI focused software products.
- Can be applied to legacy systems as well.

SbE and Acceptance Testing

The advantages of Specification by Example in terms of Acceptance testing are –

- One single illustration is used for both, detailed requirements and testing
- Progress of the project is in terms of Acceptance tests-
 - Each test is to test a behavior.
 - A test is either passing for a behavior or it is not.
 - A passing test represents that the particular behavior is completed.
 - If a project that requires 100 behaviors to be completed has 60 behaviors completed, then it is 60% finished.
- Testers switch from defect fixing to defect prevention, and they contribute to the design of the solution.
- Automation allows instant understanding of the impact of a requirement change on the solution.

Specification by Example – What it means for different Roles

The objective of Specification by Example is to promote collaboration of everyone in the team, including the customer throughout the project to deliver business value. Everyone for better understandability uses same Vocabulary.

Role	Use of SbE
Business Analyst	<ul style="list-style-type: none"> Requirements are unambiguous and without functional gaps. Developers, actually read the specifications.
Developer	<ul style="list-style-type: none"> Developers understand better, what is being developed. Development progress is tracked better by counting the specifications that have been developed correctly.
Tester	<ul style="list-style-type: none"> Testers understand better, what is being tested. Testers are involved from the beginning and have a role in the design. Testers work toward defect prevention rather than defect detection.
Everyone	<ul style="list-style-type: none"> Time is saved by identifying errors from the beginning. A quality product is produced from the beginning.

SbE – A Set of Process Patterns

As we have seen in the beginning of this chapter, Specification by Example is defined as a set of process patterns that facilitate change in software products to ensure that the right product is delivered efficiently.

The process patterns are –

- Collaborative specification
- Illustrating specifications using examples
- Refining the specification
- Automating examples
- Validating frequently
- Living documentation

Collaborative Specification

The objectives of collaborative specification are to –

- Get the various roles in a team to have a common understanding and a shared vocabulary.
- Get everyone involved in the project so that they can contribute their different perspectives about a feature.
- Ensure shared communication and ownership of the features.

These objectives are met in a specification workshop also known as the Three Amigos meeting. The Three Amigos are BA, QA and the developer. Though there are other roles in the project, these three would be responsible and accountable from definition to the delivery of the features.

During the meeting –

- The Business Analyst (BA) presents the requirements and tests for a new feature.
- The three Amigos (BA, Developer, and QA) discuss the new feature and review the specifications.
- The QA and developer also identify the missing requirements.
- The three Amigos
 - Utilize a shared model using a ubiquitous language.
 - Use domain vocabulary (A glossary is maintained if required).
 - Look for differences and conflicts.
- Do not jump to implementation details at this point.
- Reach a consensus about whether a feature was specified sufficiently.
- A shared sense of requirements and test ownership facilitates quality specifications.
- The requirements are presented as scenarios, which provide explicit, unambiguous requirements. A scenario is an example of the system's behavior from the users' perspective.

Illustrating Specification using Examples

Scenarios are specified using the **Given-When-Then** structure to create a testable specification-

Given <some precondition>
And <additional preconditions> **Optional**
When <an action/trigger occurs>
Then <some post condition>
And <additional post conditions> **Optional**

This specification is an example of a behavior of the system. It also represents an Acceptance criterion of the system.

The team discusses the examples and the feedback is incorporated until there is agreement that the examples cover the feature's expected behavior. This ensures good test coverage.

Refining the Specification

To refine a specification,

- Be precise in writing the examples. If an example turns to be complex, split it into simpler examples.
- Focus on business perspective and avoid technical details.
- Consider both positive and negative conditions.
- Adhere to the domain specific vocabulary.
- Discuss the examples with the customer.
 - Choose conversations to accomplish this.
 - Consider only those examples that the customer is interested in. This enables production of the required code only and avoids covering every possible combination, that may not be required
- To ensure that the scenario passes, all the test cases for that scenario must pass. Hence, enhance the specifications to make them testable. The test cases can include various ranges and data values (boundary and corner cases) as well as different business rules resulting in changes in data.
- Specify additional business rules such as complex calculations, data manipulation / transformation, etc.
- Include non-functional scenarios (e.g. performance, load, usability, etc.) as Specification by Example

Automating Examples

The automation layer needs to be kept very simple – just wiring of the specification to the system under test. You can use a tool for the same.

Perform testing automation using Domain Specific Language (DSL) and show a clear connection between inputs and outputs. Focus on specification, and not script. Ensure that the tests are precise, easy to understand and testable.

Validating Frequently

Include example validation in your development pipeline with every change (addition/modification). There are many techniques and tools that can (and should) be adopted to help ensure the quality of a product. They revolve around three key principles- **Test Early, Test Well** and **Test Often**.

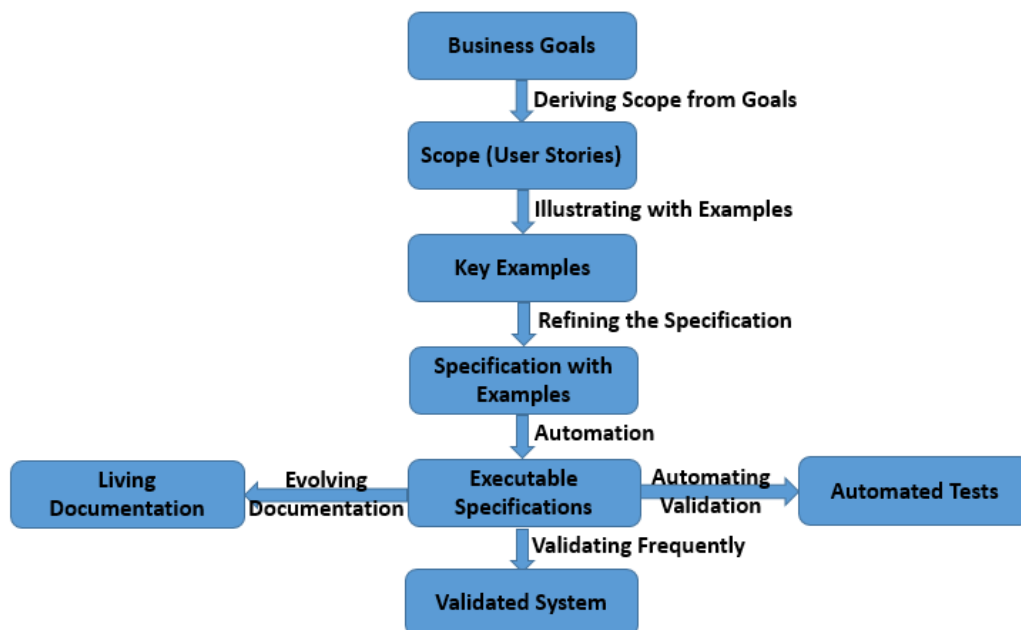
Execute the tests frequently so that you can identify the weak links. The examples representing the behaviors help track the progress and a behavior is said to be complete only after the corresponding test passes.

Living Documentation

Keep the specifications as simple and short as possible. Organize the specifications and evolve them as work progresses. Make the documentation accessible for all in the team.

Specification by Example Process Steps

The illustration shows the process steps in Specification by Example.



Anti-patterns

Anti-patterns are certain patterns in software development that is considered a bad programming practice. As opposed to design patterns, which are common approaches to common problems, which have been formalized and are generally considered a good development practice, anti-patterns are the opposite and are undesirable.

Anti-patterns give rise to various problems.

Anti-pattern	Problems
No collaboration	<ul style="list-style-type: none"> • Many assumptions • Building wrong thing • Testing wrong thing • Unaware when code is finished
Focusing on how, not what	<ul style="list-style-type: none"> • Hard to maintain tests • Hard to understand spec • Loss of interest from business representatives
Too detailed or too UI centric examples	<ul style="list-style-type: none"> • Hard to maintain tests • Hard to understand specifications • Loss of interest from business representatives
Underestimating effort required	<ul style="list-style-type: none"> • Teams think they have failed and get disappointed early

Solution to the Problems - Quality

Quality can be ensured by keeping a watch on the anti-patterns. To minimize the problems created by anti-patterns, you should –

- Get together to specify using examples.
- Clean up and improve the examples.
- Write a code, which satisfies the examples.
- Automate the examples and deploy.
- Repeat the approach for every user story.

To solve the problems due to anti-patterns means adherence to-

- Collaboration.
- Focusing on what.
- Focusing on Business.
- Be prepared.

Let us understand what each of the above mean.

Collaboration

In collaboration –

- Business people, developers and testers give input from their own perspectives.
- Automated examples prove that the team has built the correct thing.
- The process is more valuable than the tests themselves.

Focusing on what

You must focus on the question- 'what.' While focusing on 'what' –

- Do not try to cover all the possible cases.
- Do not forget to use different kind of tests.
- Keep examples as simple as possible.
- Examples should be easily understandable by the users of the system.
- Tools should not play an important part in the workshops.

Focusing on Business

To focus on the business –

- Keep specification at business intent.
- Include business in creating and reviewing specs.
- Hide all the details in the automation layer.

Be prepared

Be prepared for the following –

- Benefits are not immediately apparent, even while the team practices are changed.
- Introducing SbE is challenging.
- Requires time and investments.
- Automated testing does not come free.

Tools

Use of tools is not mandatory for Specification by Example, though in practice several tools are available. There are cases that are successful following Specification by Example even without using a tool.

The following tools support Specification by Example –

- Cucumber
- SpecFlow
- Fitnesse
- Jbehave
- Concordion
- Behat
- Jasmine
- Relish
- Speclog

5. BDD – Tools

The development teams often have a misconception that BDD is a tool framework. In reality, BDD is a development approach rather than a tool framework. However, as in the case of other development approaches, there are tools for BDD also.

Several BDD Tools are in use for different platforms and programming languages. They are-

- Cucumber (Ruby framework)
- SpecFlow (.NET framework)
- Behave (Python framework)
- JBehave (Java framework)
- JBehave Web (Java framework with Selenium integration)
- Lettuce (Python framework)
- Concordion (Java framework)
- Behat (PHP framework)
- Kahlan (PHP framework)
- DaSpec (JavaScript framework)
- Jasmine (JavaScript framework)
- Cucumber-js (JavaScript framework)
- Squish GUI Tester (BDD GUI Testing Tool for JavaScript, Python, Perl, Ruby and Tcl)
- Spock (Groovy framework)
- Yadda (Gherkin language support for frameworks such as Jasmine (JavaScript framework))

Cucumber

Cucumber is a free tool for executable specifications used globally. Cucumber lets the software development teams describe how software should behave in plain text. The text is written in a business-readable, domain-specific language and serves as documentation, automated tests and development-aid, all rolled into one format. You can use over forty different spoken languages (English, Chinese, etc.) with Cucumber.

Cucumber – Key Features

The key features of Cucumber are as follows –

- Cucumber can be used for Executable Specifications, Test Automation and Living Documentation.
- Cucumber works with Ruby, Java, NET, Flex or web applications written in any language.

- Cucumber supports more succinct Tests in Tables - similar to what FIT does.
- Cucumber has revolutionized the Software Development Life Cycle by melding requirements, automated testing and documentation into a cohesive one: plain text executable specifications that validate the software.

SpecFlow

SpecFlow is a BDD Tool for .NET Platform. SpecFlow is an open-source project. The source code is hosted on GitHub.

SpecFlow uses Gherkin Syntax for Features. The Gherkin format was introduced by Cucumber and is also used by other tools. The Gherkin language is maintained as a project on GitHub: <https://github.com/cucumber/gherkin>.

Behave

Behave is used for Python framework.

- Behave works with three types of files stored in a directory called "features"-
 - feature files with your behavior scenarios in it.
 - "steps" directory with Python step implementations for the scenarios.
 - Optionally, some environmental controls (code to run before and after steps, scenarios, features or the whole shooting match).
- Behave features are written using Gherkin (with some modifications) and are named "name.feature".
- The tags attached to a feature and scenario are available in the environment functions via the "feature" or "scenario" object passed to them. On those objects there is an attribute called "tags" which is a list of the tag names attached, in the order they are found in the features file.
- Modifications to the Gherkin Standard –
 - Behave can parse standard Gherkin files and extends Gherkin to allow lowercase step keywords because these can sometimes allow more readable feature specifications

Lettuce

Lettuce is a very simple BDD tool based on Cucumber. It can execute plain-text functional descriptions as automated tests for Python projects. Lettuce aims the most common tasks on BDD.

Concordion

Concordion is an open source tool for automating Specification by Example for Java Framework.

While the core features are simple, the [Powerful extension framework API](#) allows you to add functionality, such as using Excel spreadsheets as specifications, adding screenshots to the output, displaying logging information, etc.

Concordion lets you write the specifications in normal language using paragraphs, tables and proper punctuation and the structured Language using Given/When/Then is not necessary.

Concordion has been ported to other languages including-

- C# (Concordion.NET)
- Python (PyConcordion)
- Ruby (Ruby-Concordion)

6. BDD – BDD and Cucumber

Cucumber is a tool that supports Executable specifications, Test automation, and Living documentation.

Behavior Driven Development expands on Specification by Example. It also formalizes the Test-Driven Development best practices, in particular, the perspective of working from the outside-in. The development work is based on executable specifications.

The **key features** of executable specifications are as follows-

- Executable Specifications are –
 - Derived from examples, that represent the behaviors of the system.
 - Written with collaboration of all involved in the development, including business and stakeholders.
 - Based on acceptance criterion.
- Acceptance tests that are based on the executable specifications are automated.
- A shared, ubiquitous language is used to write the executable specifications and the automated tests such that-
 - Domain specific terminology is used throughout the development.
 - Everyone, including the customers and the stakeholders speak about the system, its requirements and its implementation, in the same way.
 - The same terms are used to discuss the system present in the requirements, design documents, code, tests, etc.
 - Anyone can read and understand a requirement and how to generate more requirements.
 - Changes can be easily accommodated.
 - Live documentation is maintained.

Cucumber helps with this process since it ties together the executable specifications with the actual code of the system and the automated acceptance tests.

The way it does this is actually designed to get the customers and developers working together. When an acceptance test passes, it means that the specification of the behavior of the system that it represents has been implemented correctly.

Typical Cucumber Acceptance Test

Consider the following example.

Feature: Sign up

- Sign up should be quick and friendly.
- Scenario: Successful sign up
 - New users should get a confirmation e-mail and be greeted personally.
 - **Given** I have chosen to sign up.
 - **When** I sign up with valid details.
 - **Then** I should receive a confirmation email.
 - **And** I should see a personalized greeting message.

From this example, we can see that-

- Acceptance tests refer to **Features**.
- Features are explained by **Scenarios**.
- Scenarios consist of **Steps**.

The specification is written in a natural language in a plain text file, but it is executable.

Working of Cucumber

Cucumber is a command line tool that processes text files containing the features looking for scenarios that can be executed against your system. Let us understand how Cucumber works.

- It makes use of a bunch of conventions about how the files are named and where they are located (the respective folders) to make it easy to get started.
- Cucumber lets you keep specifications, automated tests and documentation in the same place.
- Each scenario is a list of steps that describe the pre-conditions, actions, and post-conditions of the scenario; if each step executes without any error, the scenario is marked as passed.
- At the end of a run, Cucumber will report how many scenarios passed.
- If something fails, it provides information about what failed so that the developer can progress.

In Cucumber, **Features**, **Scenarios**, and **Steps** are written in a Language called **Gherkin**.

Gherkin is plain-text English (or one of 60+ other languages) with a structure. Gherkin is easy to learn and its structure allows you to write examples in a concise manner.

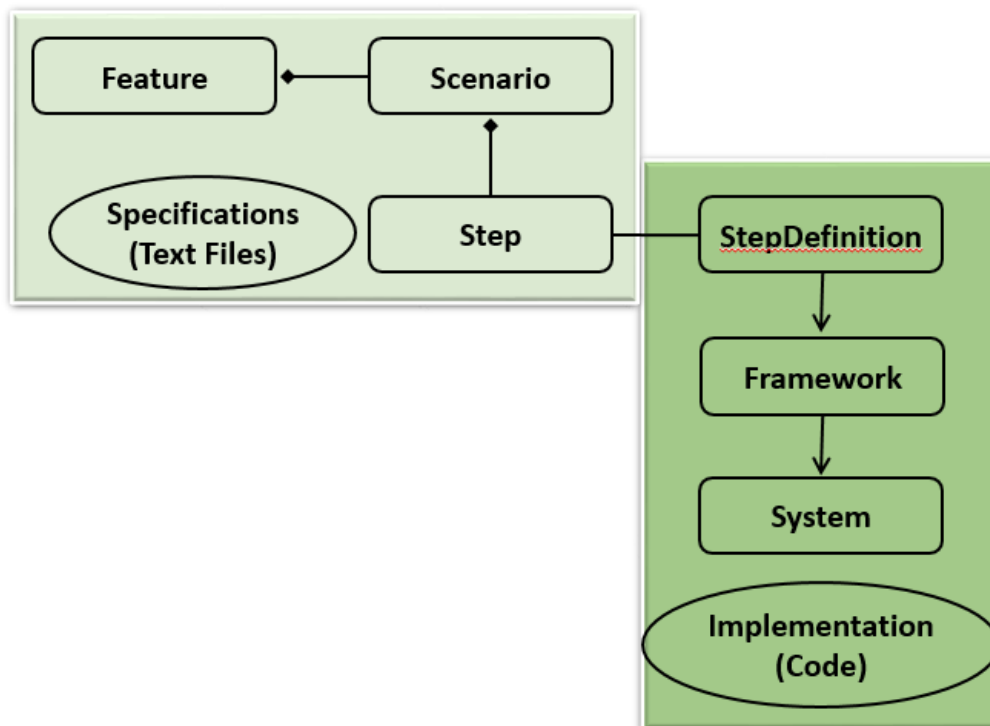
- Cucumber executes your files that contain executable specifications written in Gherkin.

- Cucumber needs Step Definitions to translate plain-text Gherkin Steps into actions that will interact with the system.
- When Cucumber executes a step in a scenario, it will look for a matching step definition to execute.
- A Step Definition is a small piece of code with a pattern attached to it.
- The pattern is used to link the Step Definition to all the matching steps, and the code is what Cucumber will execute when it sees a Gherkin step.
- Each step is accompanied by a Step Definition.
- Most steps will gather input and then delegate to a framework that is specific to your application domain in order to make calls on your framework.

Cucumber supports over a dozen different software platforms. You can choose the Cucumber implementation that works for you. Every Cucumber implementation provides the same overall functionality and they also have their own installation procedure and platform-specific functionality.



Mapping Steps and Step Definitions

The key to Cucumber is the mapping between Steps and Step Definitions.








Cucumber Implementations

Given below are Cucumber implementations.

	Ruby/JRuby
	JRuby (using Cucumber-JVM)
	Java
	Groovy
	.NET (using SpecFlow)
	JavaScript
	JavaScript (using Cucumber-JVM and Rhino)
	Clojure
	Gosu
	Lua
	PHP (using Behat)
	Jython
	C++
	Tcl

Framework Integration

Given below are Framework implementations.

	Ruby on Rails
	Selenium
	PicoContainer
	Spring Framework
	Watir

7. BDD – Gherkin

Gherkin is a language, which is used to write **Features, Scenarios, and Steps**. The purpose of Gherkin is to help us write concrete requirements.

To understand what we mean by concrete requirements, consider the following example-

Customers should be prevented from entering invalid credit card details.

Versus

If a customer enters a credit card number that is not exactly 16 digits long, when they try to submit the form, it should be redisplayed with an error message advising them of the correct number of digits.

The latter has no ambiguity and avoids errors and is much more testable.

Gherkin is designed to create requirements that are more concrete. In Gherkin, the above example looks like-

Feature: Feedback when entering invalid credit card details

Feature Definition

In user testing, we have seen many people who make mistakes...

Documentation

Background:

True for all Scenarios Below

Given I have chosen an item to buy

And I am about to enter my credit card number

Scenario: Credit card number too short

Scenario Definition

When I enter a card number that is less than 16 digits long

And all the other details are correct

Steps

And I submit the form

Then the form should be redisplayed

And I should see a message advising me of the correct number of digits

Gherkin Format and Syntax

Gherkin files are plain text Files and have the extension .feature. Each line that is not blank has to start with a Gherkin keyword, followed by any text you like. The keywords are:

- Feature
- Scenario
- Given, When, Then, And, But (Steps)
- Background
- Scenario Outline
- Examples

- """ (Doc Strings)
- | (Data Tables)
- @ (Tags)
- # (Comments)
- *

Feature

The **Feature** keyword is used to describe a software feature, and to group the related scenarios. A Feature has three basic elements-

- The keyword – Feature.
- The name of the feature, provided on the same line as the Feature keyword.
- An optional (but highly recommended) description that can span multiple lines i.e. all the text between the line containing the keyword Feature, and a line that starts with Scenario, Background, or Scenario Outline.

In addition to a name and a description, Features contain a list of scenarios or scenario outlines, and an optional background.

It is conventional to name a **.feature** file by taking the name of the Feature, converting it to lowercase and replacing the spaces with underlines. For example,

```
feedback_when_entering_invalid_credit_card_details.feature
```

In order to identify Features in your system, you can use what is known as a “feature injection template”.

In order to <meet some goal> as a <type of user> I want <a feature>

Descriptions

Some parts of Gherkin documents do not have to start with a keyword.

In the lines following a Feature, scenario, scenario outline or examples, you can write anything you like, as long as no line starts with a keyword. This is the way to include Descriptions.

Scenario

To express the behavior of your system, you attach one or more scenarios with each Feature. It is typical to see 5 to 20 scenarios per Feature to completely specify all the behaviors around that Feature.

Scenarios follows the following pattern –

- Describe an initial context

- Describe an event
- Describe an expected outcome

We start with a context, describe an action, and check the outcome. This is done with steps. Gherkin provides three keywords to describe each of the contexts, actions, and outcomes as steps.

- **Given** - Establish context
- **When** - Perform action
- **Then** - Check outcome

These keywords provide readability of the scenario.

Example

Scenario: Withdraw money from account.

- **Given** I have \$100 in my account.
- **When** I request \$20.
- **Then** \$20 should be dispensed.

If there are multiple **Given** or **When** steps underneath each other, you can use **And** or **But**. They allow you to specify scenarios in detail.

Example

Scenario: Attempt withdrawal using stolen card.

- **Given** I have \$100 in my account.
- **But** my card is invalid.
- **When** I request \$50.
- **Then** my card should not be returned.
- **And** I should be told to contact the bank.

While creating scenarios, remember 'each scenario must make sense and be able to be executed independently of any other scenario'. This means-

- You cannot have the success condition of one scenario depend on the fact that some other scenario was executed before it.
- Each scenario creates its particular context, executes one thing, and tests the result.

Such scenarios provide the following benefits-

- Tests will be simpler and easier to understand.
- You can run just a subset of your scenarios and you do not have to worry about the breaking of your test set.
- Depending on your system, you might be able to run the tests in parallel, reducing the amount of time taken to execute all of your tests.

Scenario Outline

If you have to write scenarios with several inputs or outputs, you might end up creating several scenarios that only differ by their values. The solution is to use scenario outline. To write a scenario outline,

- Variables in the scenario outline steps are marked up with < and >.
- The various values for the variables are given as examples in a table.

Example

Suppose you are writing a Feature for adding two numbers on a calculator.

Feature: Add.

Scenario Outline: Add two numbers.

Given the input "<input>"

When the calculator is run

Then the output should be "<output>"

Examples

input	output
2+2	4
98+1	99
255+390	645

A scenario outline section is always followed by one or more sections of examples, which are a container for a table. The table must have a header row corresponding to the variables in the scenario outline steps. Each of the rows below will create a new scenario, filling in the variable values

8. BDD – BDD and SpecFlow

SpecFlow is an open-source project. The source code is hosted on GitHub. The feature files used by SpecFlow to store an acceptance criterion for features (use cases, user stories) in your application are defined using the Gherkin syntax.

The Gherkin format was introduced by Cucumber and is also used by other tools. The Gherkin language is maintained as a project on GitHub: <https://github.com/cucumber/gherkin>

Feature Elements and SpecFlow

The key features of Feature elements are-

- The feature element provides a header for the feature file. The feature element includes the name and a high-level description of the corresponding feature in your application.
 - SpecFlow generates a unit test class for the feature element, with the class name derived from the name of the feature.
 - SpecFlow generates executable unit tests from the scenarios that represent acceptance criteria.
- A feature file may contain multiple scenarios used to describe the feature's acceptance tests.
 - Scenarios have a name and can consist of multiple scenario steps.
 - SpecFlow generates a unit test method for each scenario, with the method name derived from the name of the scenario.

Multiple Scenario Steps

The scenarios can have multiple scenario steps. There are three types of steps that define the preconditions, actions or verification steps, which make up the acceptance test.

- The different types of steps begin with either the **Given**, **When** or **Then** keywords respectively and subsequent steps of the same type can be linked using the **And** and **But** keywords.
- The Gherkin syntax allows any combination of these three types of steps, but a scenario usually has distinct blocks of **Given**, **When** and **Then** statements.
- Scenario steps are defined using text and can have additional table called DataTable or multi-line text called DocString arguments.
- The scenario steps are a primary way to execute any custom code to automate the application.

- SpecFlow generates a call inside the unit test method for each scenario step. The call is performed by the SpecFlow runtime that will execute the step definition matching to the scenario step.
- The matching is done at runtime, so the generated tests can be compiled and executed even if the binding is not yet implemented.
- You can include tables and multi-line arguments in scenario steps. These are used by the step definitions and are either passed as additional table or string arguments.

Tags

Tags are markers that can be assigned to features and scenarios. Assigning a tag to a feature is equivalent to assigning the tag to all scenarios in the feature file. A Tag Name with a leading @ denotes tag.

- If supported by the unit test framework, SpecFlow generates categories from the tags.
- The generated category name is the same as the tag's name, but without the leading @.
- You can filter and group the tests to be executed using these unit test categories. For example, you can tag crucial tests with @important, and then execute these tests more frequently.

Background Elements

The background language element allows specifying a common precondition for all scenarios in a feature file

- The background part of the file can contain one or more scenario steps that are executed before any other steps of the scenarios.
- SpecFlow generates a method from the background elements that is invoked from all unit tests generated for the scenarios.

Scenario Outlines

Scenario outlines can be used to define data-driven acceptance tests. The scenario outline always consists of a scenario template specification (a scenario with data placeholders using the <placeholder> syntax) and a set of examples that provide values for the placeholders

- If the unit test framework supports it, SpecFlow generates row-based tests from scenario outlines.
- Otherwise, it generates a parameterized unit-test logic method for a scenario outline and an individual unit test method for each example set.
- For better traceability, the generated unit-test method names are derived from the scenario outline title and the first value of the examples (first column of the examples table).

- It is therefore good practice to choose a unique and descriptive parameter as the first column in the example set.
- As the Gherkin syntax does require all example columns to have matching placeholders in the scenario outline, you can even introduce an arbitrary column in the example sets used to name the tests with more readability.
- SpecFlow performs the placeholder substitution as a separate phase before matching the step bindings.
- The implementation and the parameters in the step bindings are thus independent of whether they are executed through a direct scenario or a scenario outline.
- This allows you to later specify further examples in the acceptance tests without changing the step bindings.

Comments

You can add comment lines to the feature files at any place by starting the line with `#`. Be careful however, as comments in your specification can be a sign that acceptance criteria have been specified wrongly. SpecFlow ignores comment lines.