# Using acceptance test driven development to improve QA

This is an introduction/primer for:

- Acceptance Test Driven Development (ATDD)
- Behaviour Driven Development (BDD)
- Specification By Example (SBE)
- Gherkin specification syntax
- Cucumber automated testing

Most of it is adapted (ie shamelesly paraphrased/cribbed!) from the books/sites/articles in the References list.

**TL;DR**

If it looks scary or seems like wading through treacle, here's the two-minute elevator pitch:

- **Everyone benefits from clear communication and understanding of how an application should behave**
- To start the process, stakeholders agree what behaviour is expected from the application
- **Behaviour is documented in plain text specification "feature" files**
- Each feature has a set of scenarios
- Each scenario has a set of steps
- Each step defines either a pre-condition, an action, or an expected outcome
- **Everything in a feature file is written with the user vocabulary of the application**
- Scenarios can be tagged for inclusion/omission in specific test runs
- Tagging can be used for any purpose (eg business priority, test run time)
- **Everything above can be understood and done without specialist technical knowledge**
- Feature file steps are mapped to step definitions in programming code (eg Ruby)
- **Testers/developers write step definition and support code which can execute the tests**
- Test code can be written in numerous supported programming languages
- **Tests are run using a tool called Cucumber**
- Cucumber verifies measured outcome against expected behaviour
- **Outcomes are reported as passed, failed or pending (full test code not yet defined)**
- Test reports can be easily produced in various formats, including web pages for viewing in a browser
- **Cucumber tests can be run automatically within a Continuous Integration environment**
- **Incrementally updated feature files provide "living documentation" of application behaviour**

**CI**

If you don't know what Continuous Integration is, read this.

## Driving development from acceptance test definitions

*Problem*: Application and product development needs to be increasingly rapid for businesses to be competitive with today's technology

*Problem*: Rapid development teams need rapidly developed (and sufficient) specifications and tests

*Problem*: Traditional *Big Design Up Front* (BDUF) is time and resource consuming

*Problem*: Defining BDUF requirements needs specialist knowledge and skills

*Problem*: Testing against BDUF requirements needs lots of test specialists

*Problem*: Changing BDUF requirements and tracking those changes is often really tricky

*Problem*: BDUF requirements are often outdated before delivery (or even implementation)

So what are the solutions? Well, clearly for starters, traditional BDUF approaches need rethinking when rapid development and delivery is needed.

Of course, there's no silver bullet, but by embracing changing circumstances and evolving needs, and prioritising individual and team interactions, working software, and stakeholder collaboration/discussion, agile methods offer viable solutions to these problems, whilst retaining *sufficient* process rigour to assure the desired product quality.

Manual testing is well understood to be one of the most time-consuming aspects of software development for applications of any complexity. To help reduce the time needed for software verification, automated testing evolved as a key aspect of eXtreme Programming (XP). Test Driven Development (TDD) is a core XP practice, in which stakeholders and developers collaborate to write automated tests for the outcomes desired by stakeholders for their application users.

From this perspective such tests can also be seen as *acceptance tests* since they define what the software needs to do for the stakeholders - on behalf of application users - to find it *acceptable*.

It's worth noting in passing that acceptance tests are not the same as unit tests used to drive out design and coding issues. A common way of illustrating the difference is to say *unit tests ensure you build the thing right*, while *acceptance tests ensure you build the right thing*.

These ideas coined the phrase *Acceptance Test Driven Development* (ATDD), wherein conditions for acceptance are first defined which then guide the implementation teams during development.

## Driving development from behaviour

Through observation and experiences implementing TDD, Dan North realised that common misunderstandings almost always came back to the word “test”.

TDD's methods effectively ensures code works, but if they don't comprehensively describe the system behaviour they can lead to a false sense of security.

By using the word “behaviour” in place of “test” Dan found that not only did it seem to fit but also many of the questions around TDD and how to implement it magically dissolved.

What to call a test is easy – it’s a sentence describing the next behaviour in which you are interested. How much to test becomes moot – you can only describe so much behaviour in a single sentence. When a test fails, either you introduced a bug, the behaviour moved, or the test is no longer relevant.

The shift in thinking about tests to thinking about expected behaviour is now known as Behaviour Driven Development (BDD).

## How does it work?

By combining software specifications written in structured plain text with functional acceptance test conditions, BDD defines expected behaviour for user interaction scenarios associated with an application feature.

An important aspect is that the acceptance conditions are expressed in plain language using the vocabulary of the application and its anticipated users. This means they remain open and accessible to all stakeholders, not just technical implementation specialists.

Such specifications can also be localized for dozens of languages, so features can be conveniently defined to match the business needs in specific locales.

This multi-lingual plain text approach can significantly benefit the business by improving collaboration and understanding through the specification, development, testing and acceptance process, and also across geographically distributed teams.

*Gherkin* is an agreed structured text format for BDD Specifications. It uses indentation to define structure, combined with some conventions and a few simple grammar rules which are easily understood by anyone, not just technical specialists:

- Each Gherkin source file describes a single feature
- Source files have a *.feature* extension
- Line endings terminate statements
- Spaces or tabs may be used for indentation
- Most lines start with a keyword
- Comment lines are allowed anywhere, using a hash prefix (can be indented before the #)
- A set of keywords denote particular meaning for features, scenarios and steps

## Specification By Example

By using declarative language rather than imperative for the scenario steps, acceptance tests can be specified as *examples* of how an application should behave for particular scenarios, whilst remaining agnostic to specific solution dependencies (those should be covered by module integration and code unit tests).

Acceptance tests written in this style take on an extra dimension - they become *executable specifications*, such that the set of acceptance tests constitute an active, relevant specification of an application, tracking behaviour changes and updates as the tests get refined and/or refactored.

This "living documentation" aspect is another key business benefit of the BDD approach - rather than being a static document that can quickly become outdated, the test descriptions form dynamic, responsive documentation reflecting the true state of the application.

### Features and scenarios

A **Feature** is usually described with agile *user story* summary patterns:

user-orientated

```
As a [kind of user]
I want [to do something]
So that [I get some measurable benefit]
```

### benefit-orientated

```
In order to [get some measurable benefit]
As a [kind of user]
I want [to do something]
```

A key point here is that the feature is described in terms of the user experience and vocabulary of the application environment - not the technology implementation. This means the feature specifications can be easily read and understood by business stakeholders and users as well as the technical implementation teams.

Each Feature summary is elaborated with a set of scenario descriptions which cover the desired behaviours of the feature for given sets of circumstances/actions.

Although the agile user story format is a useful convention, any natural language description can be used here, so the approach is adaptable to specific needs.

A **scenario** describes the expected behaviour for particular actions given particular prior conditions, which delivers an aspect of the feature. Each step within a scenario is prefixed with a keyword. The essentials are *Given*, *When* and *Then*, any of which can be extended with *And*, *But*.

```
Given some initial context
When some action or event occurs
Then some expected outcomes should be experienced
```

*Given* puts the system into a known state before the subsequent steps. These are the *pre-conditions* for the test to be valid.

*When* describes an interaction which causes a change in state. These are most often the user interactions, but could also be automated events triggered by other state combinations.

*Then* describes the observable outcome - the expected behaviour or changed state.

If there is a need for multiple *Given*, *When* or *Then* cases, the keywords *And* and *Or* may be used to make the specification more readable, for example, this scenario:

```
Scenario: Multiple Givens
        Given one thing
            Given another thing
            Given yet another thing
        When I open my eyes
        Then I see something
            Then I don't see something else
```

Can be written as:

```
Scenario: Multiple Givens
        Given one thing
            And another thing
            And yet another thing
        When I open my eyes
         Then I see something
            But I don't see something else
```

Although this makes the definition more readable, the meaning is identical, and either version could be used for testing.

For situations where multiple scenarios with the same pattern but different test values need to be tested, Gherkin has the *Scenario Outline* syntax, with a skeleton scenario containing placeholders for sets of substitution values defined in data tables labelled with the *Examples* keyword (followed by a colon), with the first table row defining the substitution label, and a "pipe" character separating the columns:

```
Scenario Outline: Multiple sets of test data
        Given <condition>
        When <action>
        Then <outcome>

    Examples: particular related group of tests
    | condition   | action   | outcome    |
    | condition A | action A | outcome A  |
    | condition A | action B | outcome AB |
    | condition A | action C | outcome AC |
    | condition B | action A | outcome BA |
    | condition B | action B | outcome BB |
    | condition B | action C | outcome BC |

    Examples: another unrelated test set but with the same scenario pattern
    | condition   | action   | outcome    |
    | condition X | action Y | outcome Z  |
    | condition P | action Q | outcome R  |
```

Note (as shown above) that multiple data tables can be used to highlight or isolate particular test groupings. For instance, one set could relate to anonymous user interactions, whilst another is for behaviour after signing in to an application.

Multiple value substitutions can be done within a step:

```
Scenario Outline: eating
        Given there are <start> <items>
        When I eat <eat> <items>
        Then I should have <left> <items>

        Examples:
        | start | eat | left | items     |
        |  12   |  5  |  7   | apples    |
        |  20   |  5  |  15  | oranges   |
        |  15   |  3  |  12  | pears     |
```

When scenario outlines are run, each data row of the *Examples* table is translated to a single test scenario with the appropriate value substitutions. The test run is then equivalent to having each case written out "long-hand" using the normal scenario syntax.

So when running the tests, the first row of the previous example would get translated as:

```
Scenario: eating
        Given there are 12 apples
        When I eat 5 apples
        Then I should have 7 apples
```

And the last row would get translated as:

```
Scenario: eating
        Given there are 15 pears
        When I eat 3 pears
        Then I should have 12 pears
```

### Cucumber

Cucumber is a tool which bridges Gherkin format plain text specifications to test outcomes through a coding and automation layer.

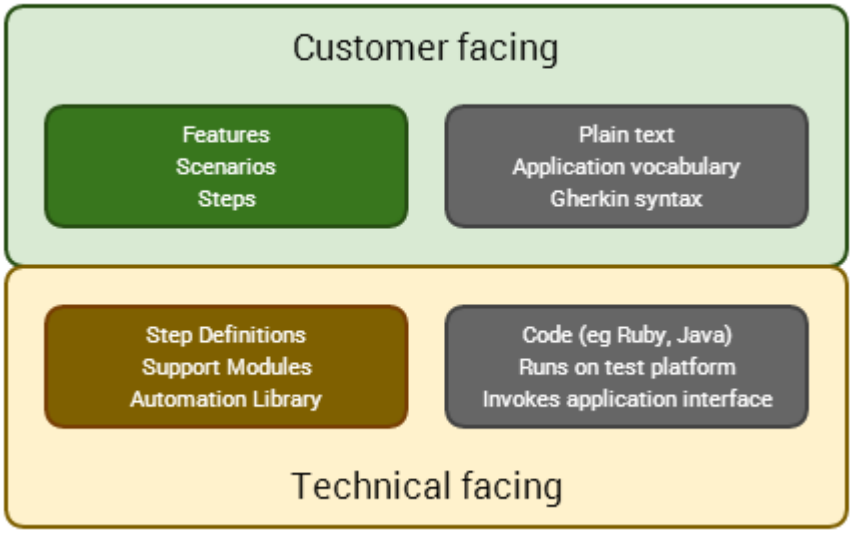
It therefore supports the automation aspects of Acceptance Test Driven Development, by reading test specifications written using the Gherkin plain text syntax, matching them to code and support libraries for particular user interaction platforms and environments, and executing that code in test runs which record observed behaviour against the expected outcomes.

Each feature scenario is essentially a list of steps for a test to work through. Cucumber maps each scenario line to *step definitions* written in an executable language (typically Ruby, but many others are supported).

In a mature application, the step definitions themselves are usually just a few lines of Ruby code which often delegate to libraries of *support code* and an *automation library* which performs the actual application interaction when tests are executed.

If the step definition code for a scenario line executes without error, Cucumber moves to the next step in the scenario. If all steps execute without error, when Cucumber gets to the end of the last step it marks the scenario as *passed*. If any steps fail, Cucumber marks the scenario as *failed* and moves to the next scenario. As the scenarios run, the results are recorded and an output report is generated.

So the BDD "test stack" using Cucumber is:




### What happens when a Cucumber test is run?

When Cucumber executes a Step within a Scenario it looks for a matching Step Definition to execute.

A Step Definition is a chunk of code matched to a particular text pattern in the feature file. The pattern links the step definition to all matching Steps, and the code is what Cucumber will execute when it sees a Gherkin Step.

Step Definitions code is placed in a file where Cucumber can find them. The location depends on which Cucumber implementation (programming language) is being used.

#### How do Step Definition work?

Consider the following Scenario fragment:

```
Scenario: Some cukes
        Given I have 48 cukes in my belly
```

The *I have 48 cukes in my belly* part of the step - the text following the *Given* keyword - will match the following Step Definition (Ruby code):

```
Given(/I have (\d+) cukes in my belly/) do |cukes|
        # Do something with the cukes
    end
```

This Step Definition uses a regular expression to match the Step source text from the .feature file. Regular expressions support "capture groups", which are strings of text matching particular bracketed patterns, such as *(d+)* in the above example, which matches any numeric digit.

Cucumber extracts any matches and automatically transforms them into an appropriate type if possible, for instance match strings containing purely digits (such as "48" above) will become numerically typed variable values which can be subsequently called within the step definition code.

Once matched, and relevant variables are extracted, the Step Definition code can implement the logic/processing required by the test condition(s), and assert true/false (or undefined) for their outcome. These assertions drive the test output report, indicating whether the test has passed or failed (or is pending full implementation).

This is very much a primer, with a lot of subtlety and detail intentionally glossed over (such as *Background* and tagging for scenarios). The following References were used to write this guide and they'll take you into that deeper detail if you're keen (and have a few hours to burn...).

## References


1. ATDD by Example: A Practical Guide to Acceptance Test-Driven Development
2. Specification by Example: How successful teams deliver the right software

3. The Cucumber Book: Behaviour-Driven Development for Testers and Developers
4. Cucumber and Cheese: A Testers Workshop
5. Instant Cucumber BDD How-to
6. Executable Specifications with Scrum: A Practical Guide to Agile Requirements Discovery
7. The Cucumber Wiki
8. Given when then

Some of the reference book links go to Safari Books Online - if you don't have an account, you can see a public preview and author/publisher information, ISBN etc.