Fork me on GitHub

## Table Of Contents

## Related Topics

Documentation overview
- Previous: Tutorial
- Next: Feature Testing Setup

## This Page

Show Source

## Quick search

Go

Enter search terms or a module, class or function name.

# Behavior Driven Development

Behavior-driven development (or BDD) is an agile software development technique that encourages collaboration between developers, QA and non-technical or business participants in a software project. It was originally named in 2003 by Dan North as a response to test-driven development (TDD), including acceptance test or customer test driven development practices as found in extreme programming. It has evolved over the last few years.

On the "Agile specifications, BDD and Testing eXchange" in November 2009 in London, Dan North gave the following definition of BDD:

> BDD is a second-generation, outside–in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.

BDD focuses on obtaining a clear understanding of desired software behavior through discussion with stakeholders. It extends TDD by writing test cases in a natural language that non-programmers can read. Behavior-driven developers use their native language in combination with the ubiquitous language of domain-driven design to describe the purpose and benefit of their code. This allows the developers to focus on why the code should be created, rather than the technical details, and minimizes translation between the technical language in which the code is written and the domain language spoken by the business, users, stakeholders, project management, etc.

## BDD practices

The practices of BDD include:

- Establishing the goals of different stakeholders required for a vision to be implemented
- Drawing out features which will achieve those goals using feature injection
- Involving stakeholders in the implementation process through outside–in software development
- Using examples to describe the behavior of the application, or of units of code
- Automating those examples to provide quick feedback and regression testing
- Using 'should' when describing the behavior of software to help clarify responsibility and allow the software's functionality to be questioned
- Using 'ensure' when describing responsibilities of software to differentiate outcomes in the scope of the code in question from side-effects of other elements of code.
- Using mocks to stand-in for collaborating modules of code which have not yet been written

## Outside–in

BDD is driven by business value; that is, the benefit to the business which accrues once the application is in production. The only way in which this benefit can be realized is through the user interface(s) to the application, usually (but not always) a GUI.

In the same way, each piece of code, starting with the UI, can be considered a stakeholder of the other modules of code which it uses. Each element of code provides some aspect of behavior which, in collaboration with the other elements, provides the application behavior.

The first piece of production code that BDD developers implement is the UI. Developers can then benefit from quick feedback as to whether the UI looks and behaves appropriately. Through code, and using principles of good design and refactoring, developers discover collaborators of the UI, and of every unit of code thereafter. This helps them adhere to the principle of YAGNI, since each piece of production code is required either by the business, or by another piece of code already written.

## The Gherkin language

The requirements of a retail application might be, "Refunded or exchanged items should be returned to stock." In BDD, a developer or QA engineer might clarify the requirements by breaking this down into specific examples. The language of the examples below is called Gherkin and is used *behave* as well as many other tools.

```
Scenario: Refunded items should be returned to stock
  Given a customer previously bought a black sweater from me
    and I currently have three black sweaters left in stock.
   When he returns the sweater for a refund
   then I should have four black sweaters in stock.,

Scenario: Replaced items should be returned to stock
  Given that a customer buys a blue garment
    and I have two blue garments in stock
    and three black garments in stock.
   When he returns the garment for a replacement in black,
   then I should have three blue garments in stock
    and two black garments in stock.
```

Each scenario is an exemplar, designed to illustrate a specific aspect of behavior of the application.

When discussing the scenarios, participants question whether the outcomes described always result from those events occurring in the given context. This can help to uncover further scenarios which clarify the requirements. For instance, a domain expert noticing that refunded items are not always returned to stock might reword the requirements as "Refunded or replaced items should be returned to stock, unless faulty.".

This in turn helps participants to pin down the scope of requirements, which leads to better estimates of how long those requirements will take to implement.

The words Given, When and Then are often used to help drive out the scenarios, but are not mandated.

These scenarios can also be automated, if an appropriate tool exists to allow automation at the UI level. If no such tool exists then it may be possible to automate at the next level in, i.e.: if an MVC design pattern has been used, the level of the Controller.

## Programmer-domain examples and behavior

The same principles of examples, using contexts, events and outcomes are used to drive development at the level of abstraction of the programmer, as opposed to the business level. For instance, the following examples describe an aspect of behavior of a list:

```
Scenario: New lists are empty
  Given a new list
   then the list should be empty.

Scenario: Lists with things in them are not empty.
  Given a new list
   when we add an object
   then the list should not be empty.
```

Both these examples are required to describe the boolean nature of a list in Python and to derive the benefit of the nature. These examples are usually automated using TDD frameworks. In BDD these examples are often encapsulated in a single method, with the name of the method being a complete description of the behavior. Both examples are required for the code to be valuable, and encapsulating them in this way makes it easy to question, remove or change the behavior.

For instance as unit tests, the above examples might become:

```
class TestList(object):
   def test_empty_list_is_false(self):
       list = []
       assertEqual(bool(list), False)

   def test_populated_list_is_true(self):
       list = []
       list.append('item')
       assertEqual(bool(list), True)
```

Sometimes the difference between the context, events and outcomes is made more explicit. For instance:

```
class TestWindow(object):
   def test_window_close(self):
       # given
       window = gui.Window("My Window")
       frame = gui.Frame(window)

       # When
       window.close()

       # Then
       assert_(not frame.isVisible())
```

However the example is phrased, the effect describes the behavior of the code in question. For instance, from the examples above one can derive:

- lists should know when they are empty
- window.close() should cause contents to stop being visible

The description is intended to be useful if the test fails, and to provide documentation of the code's behavior. Once the examples have been written they are then run and the code implemented to make them work in the same way as TDD. The examples then become part of the suite of regression tests.

## Using mocks

BDD proponents claim that the use of "should" and "ensureThat" in BDD examples encourages developers to question whether the responsibilities they're assigning to their classes are appropriate, or whether they can be delegated or moved to another class entirely. Practitioners use an object which is simpler than the collaborating code, and provides the same interface but more predictable behavior. This is injected into the code which needs it, and examples of that code's behavior are written using this object instead of the production version.

These objects can either be created by hand, or created using a mocking framework such as mock.

Questioning responsibilities in this way, and using mocks to fulfill the required roles of collaborating classes, encourages the use of Role-based Interfaces. It also helps to keep the classes small and loosely coupled.

## Acknowledgement

This text is partially taken from the wikipedia text on Behavior Driven Development with modifications where appropriate to be more specific to *behave* and Python.