**Gherkin:**

Is a **domain specific language** for **writing acceptance criteria** that has five main statements:

* Feature – high level description of a software feature.
* Scenario / Example – a label for the behavior and consists of list of steps.
* Given – the beginning stat of the scenario. (step)
* When – a specific action that the user takes. (step)
* Then – a testable outcome, usually caused by the action in When. (step)
* And – this continues any of the other three operators. (steps)
* Scenario Outline / Scenario Template

*Writing automated unit tests for the software based upon the acceptance criteria.*

*Writing acceptance criteria in Gherkin form is a great way to define and agree on what ‘done’ means for the features we build.*

**Feature files:** are **executable specifications** and it is **an entry point** to the **Cucumber tests**, consists of Scenario or group of Scenarios. Feature files are like virtual checklist for making sure that software works.

**Cucumber**: is a tool that allows us to create automated software tests in an easy to write, easy to read way.

Translating **Gherkin scenarios** to a **code** uses a technology called **Regular Expressions** (Regex). Cucumber **recognizes Gherkin’s keyword** defined in **scenarios (phrases)** and **translates into a code** by something called Step Definitions.

The English Gherkin scenarios we write can be automatically translated into testable automated code.

**Step Definitions:**

**Each step** in a **scenario** should have **step definition** so that the automated test suite knows how to translate our English into code, Translated English scenario steps into code that we can run.

How Gherkin + Cucumber implementation works and how building a library of well written Gherkin scenarios pave the way for automated testing.

**Behavior Driven Development (BDD)**

In Agile environments, **Behavior Driven Development (BDD)** plays a vital role because it strongly encourages the use of Agile methodologies during the development and testing.

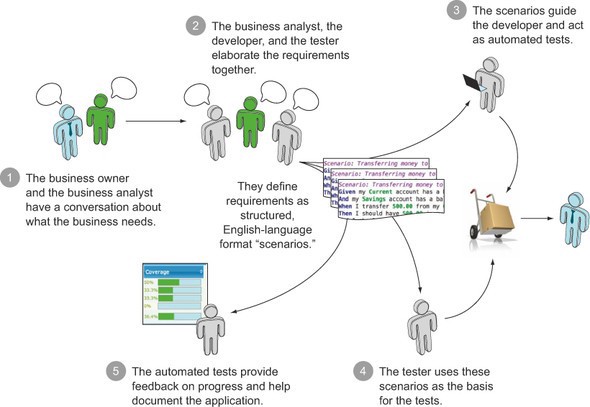
BDD brings customers, end-users, BAs, QAs, and SEs of the software product into one table for effective sharing of knowledge on the system and its testing requirements.

BDD is a method carefully developed while keeping the principles of Agile Manifesto in mind.

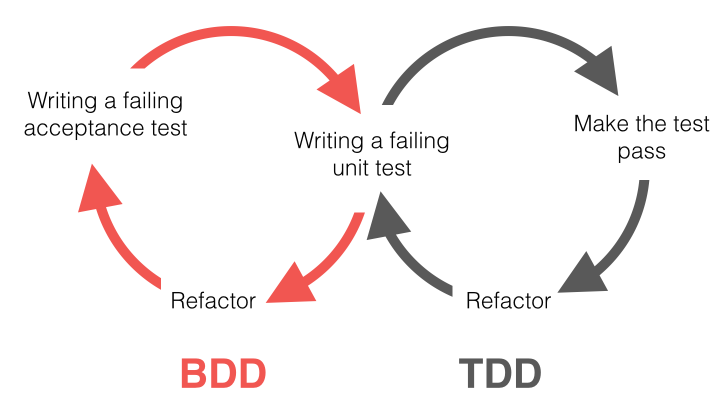
**Three Amigos**

The people in charge of defining the requirements (business analysts / agile product owners) sit down with programmers and testers and discuss **features (**similar to **agile stories)** to be implemented.

* The **business person** specifies behaviors they want to see in the system.
* The **developer** asks questions based on their understanding of the system, while also writing down additional behaviors needed from a development perspective.
* The **testers** decides on which system behaviors the acceptance tests will be written.



These **three amigos (business persons, developers, testers)**come up with **examples** of how the software should behave, and write them down as **Cucumber Features and Scenarios**. Thereafter the software development happens following the principles of **ATDD (Acceptance Test Driven Development**) and **TDD (Test Driven Development**).



**Myths on Cucumber BDD**

There are people who have misunderstood this concept of BDD and Cucumber features. Some of them use Cucumber as a testing tool and BDD as a testing method. Some write Cucumber features after the development of their software product in order to reflect its behavior.

Although everyone has the freedom to use these tools as they wish, this crowd is missing the bigger purpose of this collaboration tool, **creating a shared understanding among three amigos by discussing examples on expected software product**.

*Cucumber is first and foremost a collaboration tool that aims to bring a common understanding to software teams — across roles.*

Therefore, it should be noted that **BDD tools like Cucumber are made to drive the entire life cycle of software implementation in more collaborative manner.**

*Cucumber features should drive the implementation, not reflect it.*

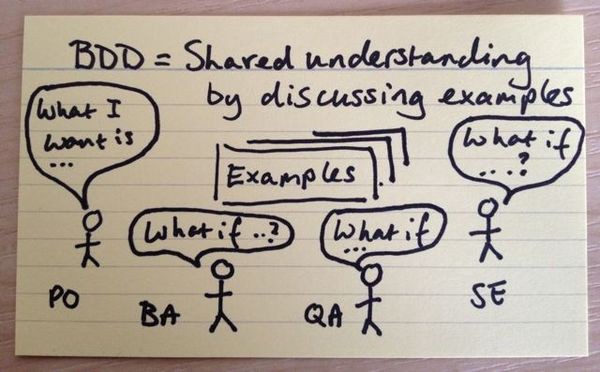
*Cucumber is not a tool for testing software. It is a tool for testing people’s understanding of how software (yet to be written) should behave.*

The biggest advantage of **BDD approach for software development** might be that **they describe a set of functions that a user expects from a system in a very concrete and direct manner**. The sum of these behaviors essentially document a contract with the user/client. If any of the tests fail, this contract is not upheld.

**Let’s simulate BDD:**

Millennials love building stuff than reading docs. So let’s simulate the stages of BDD in a very high level manner and create a dummy project with Cucumber BDD to understand the workflow. Note that this is just for your understanding purposes :-)

Assume that our product is a web-based **HRM software** and we are gonna discuss the profile login and update features with ***specification by example*** method.



**Writing Executable Specifications with Examples**

This is the most important stage of BDD. Three amigos ( business persons, developers, testers) get together and identify the expected behavior of our product by discussing **examples**. We can use **feature mapping** approach to effectively analyse and elaborate the product behavior.

**Feature Mapping:**

**Feature Mapping** is a simple collaborative practice to help teams write great executable specifications.

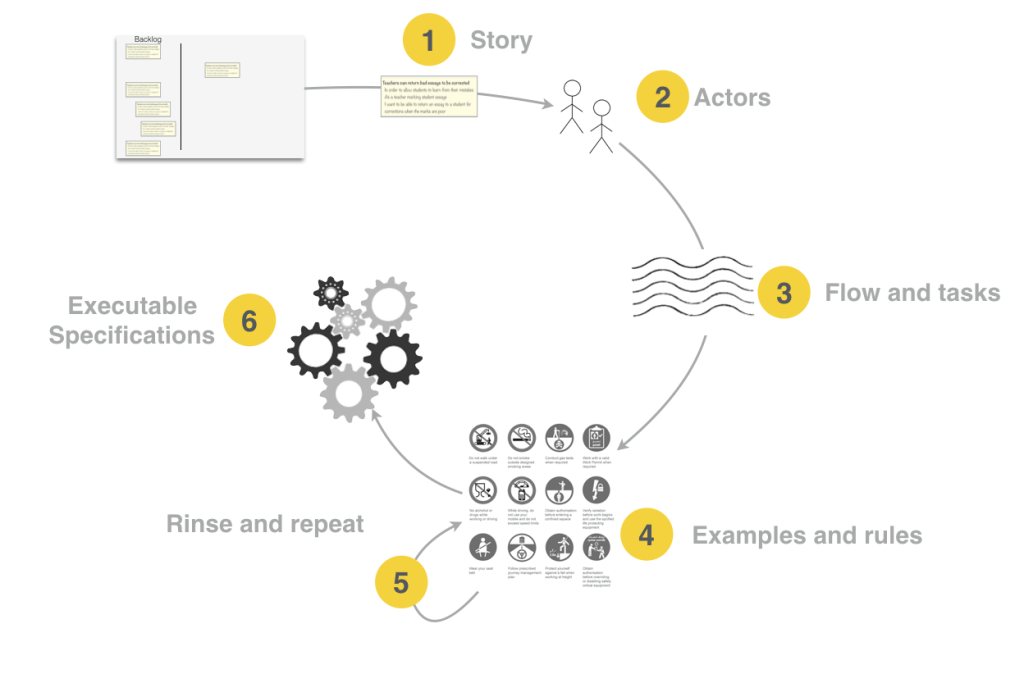
Feature Mapping draws on [Story Mapping](http://jpattonassociates.com/user-story-mapping/) or [Example Mapping](https://cucumber.io/blog/2015/12/08/example-mapping-introduction), and other techniques.

In a nutshell, the process goes something like this:

1. **Define a feature** or **story**, or pick one from the backlog.
2. **Understand** what**actors** are involved in the story.
3. **Break the feature into tasks** **to identify** the **main flows**.
4. **Identify examples** that illustrate a principle or variant flow. Ask questions like: “But what if…”; “what else could lead to this outcome”; “what other outcomes might happen”;

**A**nd use the answers to create new examples. **Use rules** **to explain and give context to your examples**.

1. **Rinse and repeat** for other rules and examples.
2. **Create Executable Specifications**: automate the main high-level flows with pending steps.



**Feature Mapping Cycle:**

Assume that the discussions of our three amigos are happening now. They write the outcome of their discussions as high level features and scenarios in a business readable grammar. If you are familiar with agile, this process is very similar to the well-known agile story writing. The following are the features we discuss. Notice that they are very**high-level** and **abstract**.

**Feature:** Login Profile  
 As an employee of the company  
 I want to login my employee profile using my credentials  
 In order to collaborate with my colleagues

**Feature:** Update Profile  
 As an employee of the company  
 I want to be able to update my name, projects, email, and phone numbers on my profile

In order to share my contact information with my colleagues

In our case, notice that there is only**one actor**— the employee.

Now we are ready to **break feature into tasks and scenarios**.

**Feature:** Login Profile  
 **Scenario:** Successful login  
 **Scenario:** Failed login using wrong credentials

**Feature:** Update Profile  
 **Scenario:** Update name  
 **Scenario:** Add new projects

To execute scenarios, we need to**establish the steps and rules** involved in each scenario. For an example, let’s consider ***successful login scenario***.

Before log into profile, employee must first visit the company home page. This is a precondition, i.e. **background scenario** which helps to successfully perform our expected scenario, ***successful login***. So let’s identify its’ steps.

**Background:** User navigates to Company home page

**Steps:** 1) Go to company home page   
 2) See login option (this is a consequence of above step)

**Scenario:** Successful login

**Steps:** 1) Enter correct credentials   
 2) Login to account  
 3) See welcome message (this is a consequence of above step)

Now comes a tricky part, the realization of above features and scenarios. We need to get the inputs of developers and testers and convert above story-like scenarios to executable steps in **Given/When/Then** format of **Gherkin language**. Also, it should be noted that the business persons must agree with these steps.

**Background:** User navigates to Company home page

**Given** I am on the "**Company home**" page on URL "**www.mycomany.com**"  
 **Then** I should see "**Log In as Employee**" message

**Scenario:** Successful login  
 **When** I fill in "**Username**" with "**Test**"  
 **And** I fill in "**Password**" with "**123**"  
 **And** I click on the "**Log In**" button  
 **Then** I am on the "**My profile**" page on URL "**www.mycompany.com/myprofile**"  
 **And** I should see "**Welcome to your profile**" message  
 **And** I should see the "**Log out**" button

**NOTE**: The above scenario is written to serve as a simple example and it is not the best approach to write scenarios. In production, always make sure that your scenarios are not tightly coupled with your tests. Your BDD scenarios should change only when the requirement changes, not when the the implementation changes (i.e. your BDD scenarios must drive the implementation, not the other way around).

# **Feature Mapping: A Simpler Path From Stories to Executable Acceptance Criteria:**

**Writing good acceptance criteria** is one of the keys to **effective software delivery**.

An **Executable Specification is a Definition of Done** that you can **run as a test.** In **Behavior Driven Development (BDD)**, we refer to **acceptance criteria as “executable specifications.**” Executable Specifications are meant to be clear, unambiguous, written in business terms, and easy to automate. Each acceptance criteria is a concrete example of how a user interacts with the system to achieve some business goal.

The most well-known **format for BDD acceptance criteria** uses the “**Given-When-Then**” structure: This Scenario gives a nice illustration of one particular path through the application workflow.

**We Start with the Story**

Feature Mapping starts with a feature or story, typically from your existing product backlog. Suppose we have the following story for our school application:

Feature: Teachers can return bad essays to be corrected

In order to allow students to learn from their mistakes

As a teacher marking student essays

I want to be able to return an essay to a student for corrections when

the marks are poor

As a rule, good acceptance criteria avoid mentioning user interfaces, field formats, buttons and so on. Mockups or wireframes are a much better place to document these requirements.

### We Understand the Actors

Most stories involve at least one, and sometimes several, actors. Understanding which actors are involved helps us reason about the tasks they need to perform to achieve the story goals. In our case, we can identify two actors:

* the student who submits the essay (let’s call this actor Stu)
* and the teacher who marks the essay (Tess)

**We Break the Feature Into Tasks or Steps**

Next, we break the feature into steps or tasks. How does the actor (or actors) interact with the application to achieve the goal of this story? What tasks do they need to perform? If you were manually demonstrating how this feature worked, or that it worked, what are the steps you would need to do? This is like Story Mapping, but we are focusing on a particular feature or story, rather than trying to get a high-level view of the application.

In the story shown above (*“Teachers can return bad essays to be corrected”*), we might identify five main tasks:

* The student submits his essay for marking.
* The teacher opens the essay.
* The teacher records marks for each category.
* The teacher returns the essay to the student for correction.

### Examples, Rules, and Questions

#### **Finding Examples**

Once we have an idea of the tasks or steps involved in achieving the story goal, we talk through concrete examples of these steps. Each example illustrates a different flow through the steps. For our first example, we explore what happens when Stu submits a great essay:

* Stu got 9 in every category so his essay is saved in the final results

If we map this out into the various tasks we identified earlier, we would get something like this:

* The teacher saves the final results.

**Working with Negative Cases**

The examples we’ve seen so far could be described as “happy-day cases” - smooth flows through the story. But our acceptance criteria should also describe negative scenarios, especially if they are important to the business. Negative scenarios help understand the positive scenarios better and help flush out incorrect assumptions or missing details.

Acceptance criteria should record both positive and negative scenarios. However, not *all* negative scenarios are useful. Field validation rules are fine if the rules relate to business requirements. But a scenario checking for badly-formatted dates or numbers would normally be reserved for unit testing.

For example, the rule that a teacher can return an essay if there is a mark of 6 or under begs the question: can a teacher return an essay if the marks are good? This would justify a separate scenario:

* Tess records 9/9/9 for Stu’s essay and tries to return it, but she is not allowed to.

### Adding Extra Details

Sometimes we want to add some extra information. We might want to add the error message that should be displayed if an invalid mark is entered; or we might want to add a table of test data describing different variations of the same scenario (for example, different mark weightings for different subjects). I generally put these details on the task card (if there is room), on the back of the task card, or sometimes on a card underneath the main task card. The main thing is to have them handy so that we can refer to it later when we automate.

## **From Feature Mapping to Test Automation**

One of the nice things about this approach is that we can start automation immediately. Each example maps to a clear sequence of business-level tasks, which in turn are easy to automate.

### Feature Mapping with Cucumber

In Cucumber, for example, we can map the steps more-or-less directly to steps in the Cucumber scenario:

Scenario: Returning an essay to the student for correction

A teacher can return an essay to the student to be corrected if any mark is 6 or less

Given that Stuart has submitted an essay on 'Politics 101' to be marked

And that Tess has opened the essay

When Tess records the following marks:

| Spelling | Reasoning | Relevance |

| 6 | 6 | 6 |

And Tess returns the essay to Stuart to be corrected

Then Stuart should put the 'Politics 101' essay in his Pending Correction list

### Feature Mapping with ScreenPlay in JUnit

Using the [Screenplay pattern](https://www.infoq.com/articles/Beyond-Page-Objects-Test-Automation-Serenity-Screenplay) in Java with [Serenity BDD](http://serenity-bdd.info/#/), with either JUnit or Cucumber, we could automate these steps like this:

Actor tess = Actor.named("Tess").whoCan(MarkPapers);

Actor stuart = Actor.named("Stuart").whoCan(SubmitPapers);

givenThat(stuart).wasAbleTo(

Submit.anEssayAbout("Politics 101")

);

andThat(tess).wasAbleTo(

ReviewTheEssay.from(stuart).about("Politics 101")

);

when(tess).attemptsTo(

RecordMarks(of(6).in(Spelling), of(6)).in(Reasoning), of(6).in(Relevance)),

ReturnTheEssay.forCorrections()

);

then(stuart).should(

seeThat(HisEssays.thatAre(PendingCorrection), contains("Politics 101"))

);

Notice how clearly the tasks map to steps in Cucumber and Java. The structure of the automated tests maps closely to the business flow in the examples. This makes both the reporting clearer and the code easier to understand.

At this point, you can automate minimal implementations for your tasks, and you will have a pending executable specification which will act as a starting point for your test automation efforts.