Gherkin Behavior Driven Testing: Hype or Not?

By Matthew Heusser Matt@xndev.com for Subject7

Behavior Driven Development was a mind hack.

Dan North, the creator, was teaching unit testing to programmers and noticed that programmers found testing … well … boring - which made the idea of driving development with tests even less interesting. Yet what those tests were doing was really defining behavior, or setting expectations. A test written before code is created is, after all, a sort of executable specification.

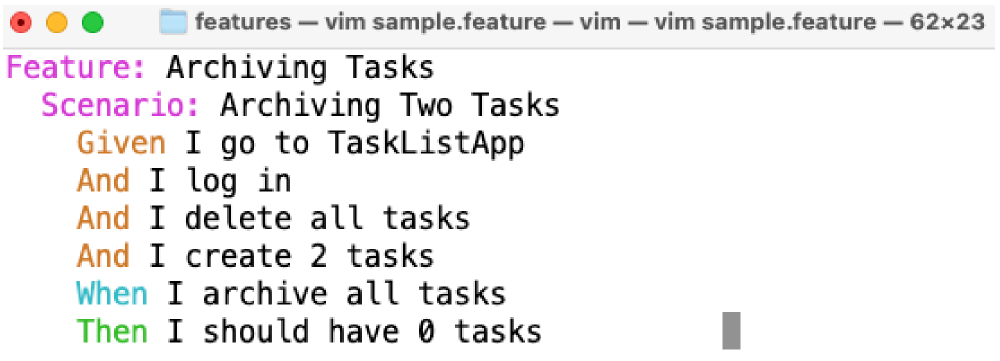
Dan found that by changing his language, from "test" to "behavior," or "expectation," he could get the programmers interested again. Years later, Dan, Chris Matts, and Liz Keogh took a common template for requirements and turned it into a template for a real example: Given, When, Then. This template language became known as Gherkin, known as a "Plain English" way to gather requirements which would later become executable examples. Since then, BDD has captured the imagination of both executives and programmers.

The question is - Does it work?

More specifically, *is* Gherkin plain English? Does gathering requirements in Gherkin really save time, make testing easier, or in other ways live up to its hype? To answer these questions, I'll go deeper than just talking about it, but instead work a real example of Given/When/Then. I'll talk about the pros and cons of Gherkin, compare it to other approaches, and talk about when it could work.

A Simple Gherkin Example

Our intention here is to create a real computer program, called TaskList, that stores a list of tasks. Early on, the team gathers requirements in English, collaboratively. Here's a sample features file for the "archive" feature:


features — vim sample.feature — vim — vim sample.feature — 62×23

```
Feature: Archiving Tasks
  Scenario: Archiving Two Tasks
    Given I go to TaskListApp
    And I log in
    And I delete all tasks
    And I create 2 tasks
    When I archive all tasks
    Then I should have 0 tasks
```

Notice that we can put many scenarios in one feature file, and many feature files in a single application. The scenario does have a very English feel. Testers could cut and paste this code, change numbers, change the order, and so on. As a tester, one of my first scenarios would be to create two tasks and make sure two tasks appear, then I would explore the difference between deleting tasks and archiving them. There is a lot here, but at this level, Gherkin does seem like a nice, easy, codeless way to express requirements.

The thing is, most requirements *are* expressed in Plain English. Plain English is just sort of how we express requirements. There is no magic here. The magic for Gherkin is that we can transform these precisely worded documents into tests.

To do that though, we have to actually write some code, called a "step definition." Step definitions are the actual computer code that will implement these ideas. To create a step definition, I saved my feature file, went to the command line, and typed in "cucumber features/sample.feature." That ran the sample feature, which had half a dozen steps that are not implemented, and generated a report creating my sample step file in Ruby, which are the actual functions that implement "I log in" and so on. Here's the generated code; feel free to skim:

```
Given('I go to TaskListApp') do
  pending # Write code here that turns the phrase above into concrete actions
end

Given('I log in') do
  pending # Write code here that turns the phrase above into concrete actions
end

Given('I delete all tasks') do
  pending # Write code here that turns the phrase above into concrete actions
end

Given('I create {int} tasks') do |int|
# Given('I create {float} tasks') do |float|
  pending # Write code here that turns the phrase above into concrete actions
end

When('I archive all tasks') do
  pending # Write code here that turns the phrase above into concrete actions
end

Then('I should have {int} tasks') do |int|
# Then('I should have {float} tasks') do |float|
  pending # Write code here that turns the phrase above into concrete actions
End
```

Someone has to go and *write* these functions, to implement the "step definitions." That is when the step "I log in" occurs, someone has to write a method to:

Go to the homepage
Figure out if the user is logged in (if yes, log out)
Log in (should username and password be passed in? Have defaults)
Probably verify that the environment is now logged in

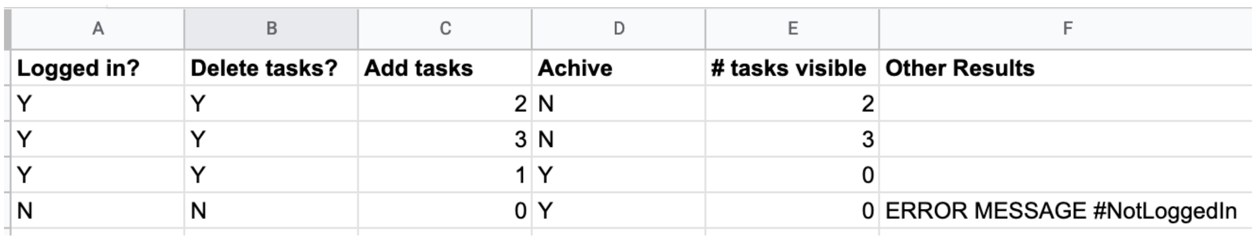
That needs to be done with code. Pure, old fashioned, code. If the program is a web application, the programmer will probably have to write Selenium code. Once that is done, the less-technical people on the team can cut and paste scenarios, move things around, change numbers, change the order of things, even invent new commands. Those commands still need to be coded and changed when the user interface changes. All the usual things we have learned about GUI test automation apply. The programmers will want to separate the locators of the user interface, so when they change, they will only change in one place, a principle called Don't Repeat Yourself, or DRY. Cucumber does not do anything to enforce DRY, leaving it up to the programmers to create something beautiful, or without skill, a big ball of mud.

What Cucumber does is create an incredibly powerful language that resembles English and follows a template style. That means anyone can understand and create in it. The difference between feature files and step definitions provides a *separation* between those Gherkin tests, which are English-like, and the implementation, done by a computer programmer. With Gherkin, it is at least possible that customers could review tests, be involved in them, and agree that, yes, that is what the computer should do. Doing that collaboratively, as a team, all on one call, before the code is created, is the heart of what has come to be known as Behavior Driven Development and Specification by Example. The technology also provides a way to share functionality between features, called the support directory.

So that's the technology. Let's talk about the benefits, drawbacks, and risks to see if it is a fit for your organization.

Pros and Cons of Gherkin, Cucumber, and Behavior Driven Development (BDD)

Gherkin isn't the only way to model tests. It is easy enough to create a spreadsheet, like the one below. Once the spreadsheet exists, any programmer can automate the code very similar to the step definitions.



A	B	C	D	E	F
Logged in?	**Delete tasks?**	**Add tasks**	**Achive**	**# tasks visible**	**Other Results**
Y	Y	2	N	2	
Y	Y	3	N	3	
Y	Y	1	Y	0	
N	N	0	Y	0	ERROR MESSAGE #NotLoggedIn

The test above could scale to dozens of tests on a single page. Any Gherkin reader comes in danger of "losing the plot" after about a half dozen tests. Specifying the behavior of an entire system in Gherkin will likely be several hundred tests, which leads to an overwhelming amount of text. Gherkin does provide a mechanism to combine many similar tests into one, called the data table, but that misses the real power of the language itself.

Fundamentally, spreadsheets, like the example above, expect a pre-determined workflow. Gherkin is event-driven, it allows the user to change the order of events to create a different workflow. By changing the order of the text, the Gherkin author can have the delete happen at the end, or the archive happen at the beginning. This adds a great bit of variety to the tests while allowing for re-use of the step definitions. The spreadsheet above could work for a user interface, but it is more likely suited for a batch program, where the same order of operations will happen again and again.

The English-like nature of BDD makes it easier to get less-technical customers, including the stakeholders that pay the bills for the project, involved in the details of what the software should do. It allows the requirement creators to focus on workflow and outcomes, without bogging them down in hard details of software engineering. As such, Gherkin is a great step forward for clarifying what to build, how it should work, and having a first pass at functionality. They are not quite as useful as acceptance tests, because if they pass that does not mean the software should be accepted. Still, if they fail, the software certainly fails -- what testing expert Michael Bolton calls "Rejection checks."

Finally, one risk that people don't look into often enough for software development is the risk that the idea will not transfer. That is, you could hire a trainer to deliver a three day course, then flip over to the new technology. When you bring the trainer back six months later, what they find has no meaningful relationship to the idea they were trying to explain. This is probably the case in software more often than not. Today, the company is less likely to hire an in-person trainer and more likely to read ideas on the internet, which has no opportunity for feedback and an even lower transference rate.

What happens most of the time with BDD is continued conversations, and that lead to given / when / then that are overly complex, thus never implemented. Without a connection back to the actual software, these eventually become stale. In my experience, that is the greatest risk with BDD, which is often still a win for the requirements process and the testers, as they now have some clear expectations of how to check whether the software is doing what is expected. These risks increase as the software increases in team size, complexity, and time.

The Bottom Line on Gherkin and BDD

The best-fit project for Gherkin will probably be a small one, where the entire team can be convinced to try the method, from requirements to executable tests, for some period of time. That should probably be an event-driven project with a graphical user interface, where users can bounce around the screen, get the software into an odd state, and the team can document what should happen in that state. The team should have a plan for who will write the "plumbing" code to implement the tests (it will be extra work), along with who will be responsible for the English-like feature files, and who will add the Cucumber test run to continuous integration.

With all that in place, it is time for your team to try BDD for three months, then step back and see what you've learned. You might not end up with test automation, you might just have learned a thing or two about human interaction. The process of trying to create step definitions and separate them from Gherkin will make better programmers, and even writing the Gherkin will help the less-technical people appreciate programming.

If the description above is not you, then BDD might not be a fit. Feel free to try, just tread lightly. Be leery of creating systems that are later abandoned … they tend to create delays and add drag to the delivery team.

Good luck. Let us know how it goes; we'd love to hear from you…