BDD 101: A Comprehensive Guide to Behavior Driven Development

SMARTBEAR

Content

What is BDD?

Behavior-driven development, or BDD, helps business and technical teams better communicate to avoid waste. Using examples witten in a domain-specific language, or DSL, BDD enables non-technical users to express software behavior and expected outcomes. Technical users convert these examples into executable specifications that help ensure that their code meets business objectives outlined in user stories and acceptance criteria.

Let's take a look at how the BDD process works from a high level to better understand the terminology and concepts before diving deeper into the benefits and implementation.

The Two-Part Process

Behavior-driven development is a two-part process consisting of a deliberate discovery phase and a testing phase. While most developers are familiar with testing, from test-driven development, deliberate discovery is often a new and profound experience.

Deliberate Discovery

The deliberate discovery phase brings together key stakeholders to have conversations and come up with concrete examples. Product owners, business analysts, domain experts, users, developers, UX designers, testers, ops engineers, and others meet to come up with concrete examples of user stories and acceptance criteria. These examples ultimately become the automated tests and living documentation showing how the software behaves.

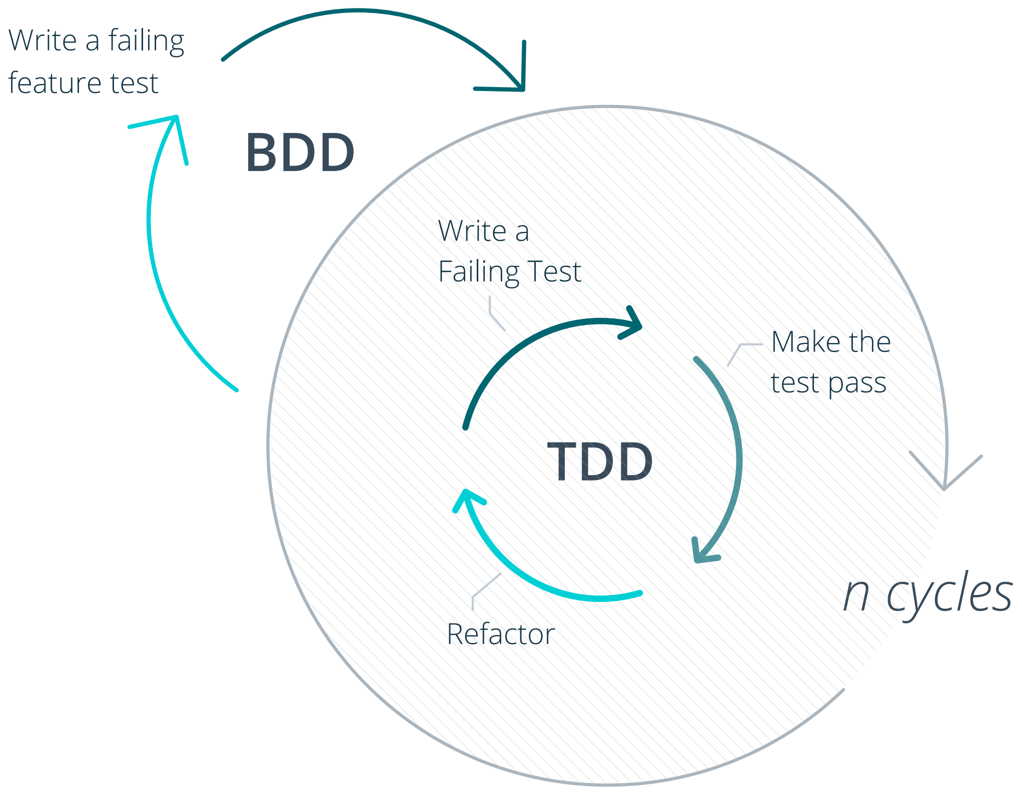
Testing Phase

The testing phase is where concrete examples become software. Using a BDD framework and DSL, the natural language used to describe the concrete examples in the deliberate discovery phase is converted into an executable specification. Developers write code to make these tests pass, which eliminates rework caused by vague requirements, slow feedback cycles, and other problems that commonly arise in non-BDD scenarios.

These BDD tests, focused on business-facing features, work in conjunction with TDD tests, which focus on lower level implementation details.

Where It Fits in Development

In the Agile development cycle, BDD processes usually take place when you'd normally come up with acceptance criteria. The process involves converting



these acceptance criteria into concrete examples and automated tests before putting them into the development pipeline. Developers can then use the automated tests to drive their development process rather than trying to write code based on potentially vague acceptance criteria.

BDD Frameworks & DSLs

There are many different behavior-driven development automation frameworks and domain-specific languages—and more are appearing every day. While all BDD automation frameworks follow the same principles, different frameworks target different languages and platforms. It's important to consider both the maturity of the BDD framework and its suitability for your application when choosing between frameworks for your organization.

BDD Automation Frameworks

- **Cucumber**: Cucumber is the original BDD automation framework that began with Ruby (a pioneer of TDD and BDD) and expanded into other languages.
- **SpecFlow**: SpecFlow is "Cucumber for .NET" and has become the most popular BDD automation framework for Microsoft's .NET framework.

- **Jasmine**: Jasmine is the most popular BDD automation framework for JavaScript applications, which are becoming increasingly popular in startups.
- **Easyb**: Easyb is a BDD automation framework designed for Java, which is widely used across enterprise applications.
- **Behave**: Behave is a BDD automation framework designed for Python, which is commonly used for data-driven and scientific applications.
- **Behat**: Behat is a BDD automation framework designed for PHP 5.3+, which is used across many legacy web applications.

Domain-Specific Languages (DSLs)

- **Gherkin**: Gherkin is by far the most popular DSL used by BDD automation frameworks.

Common Misconceptions

"BDD is just a testing framework."

Behavior-driven development is not a testing technique, it's a development strategy. It's the process of thinking about the user experience and application programming interface, or API, before writing a single line of code. Automated tests are more of a byproduct of BDD than the purpose of using the strategy. This means that you should never be writing tests after writing code since the tests aren't driving the implementation in that case.

"BDD introduces unnecessary complexity."

Many developers already use test-driven development, which includes unit and integration tests. These processes may already test user interfaces with automation frameworks, like Selenium and Capybara. Some developers see business readable language as an unnecessary step that slows down the test suite. But the two are completely different: BDD is about collaboration, not testing automation.

"BDD will take longer and delay projects."

BDD introduces some extra meetings to ensure that business and technical users are on the same page, but these meetings don't have to be very long and they can help your team be much more productive over the long-term. These meetings also ensure that developers aren't wasting their time writing the wrong code. We will see in the next section how BDD can help deliver real benefits with a minimal time commitment.

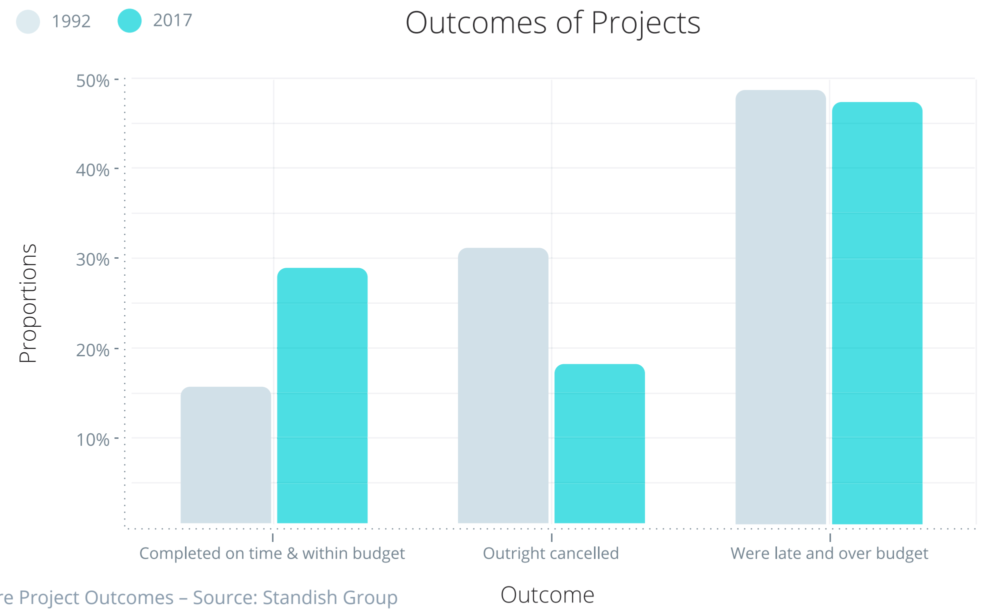
Why Use Behavior Driven Development?

How many times have you written acceptance criteria that were not properly implemented by a developer? How many times have you had a conversion with a developer only to find that you both had different ideas after parting ways? Behavior-driven development attempts to resolve these common challenges that arise during software development to make it a much smoother process for everyone involved in the organization.

Let's take a look at why software development is so challenging, why traditional Agile methodologies aren't enough, and the case for using BDD in your organization.

Software Development is Hard

Less than one-third (30%) of software projects are completed on time and on budget, according to the Standish Group, and another one-fifth of projects are canceled before completion. In other words, only about half of software projects could actually be considered "successful" in being both on time and on budget. There are many reasons that a



Software Project Outcomes – Source: Standish Group project may go over the original time and budget, but most problems boil down to communication.

Software Project Outcomes – Source: Standish Group

The good news is that these outcomes have improved over time. In fact, the number of software projects delivered on time and on budget roughly doubled between 1992 and 2017, while the number of projects that were canceled nearly halved. The number of late projects that went over-budget, however, remained roughly even over the period. These trends suggest that at least some new software development techniques are working.

Agile development is perhaps the most influential software development methodology to surface over the past decade. While "Agile" is a very broad topic, the underlying goals are to bring business and technical teams closer together, release software in more frequent cycles, and adapt to evolving customer needs rather than sticking to a plan. The result is better software projects that more closely match expectations—resulting in fewer canceled projects.

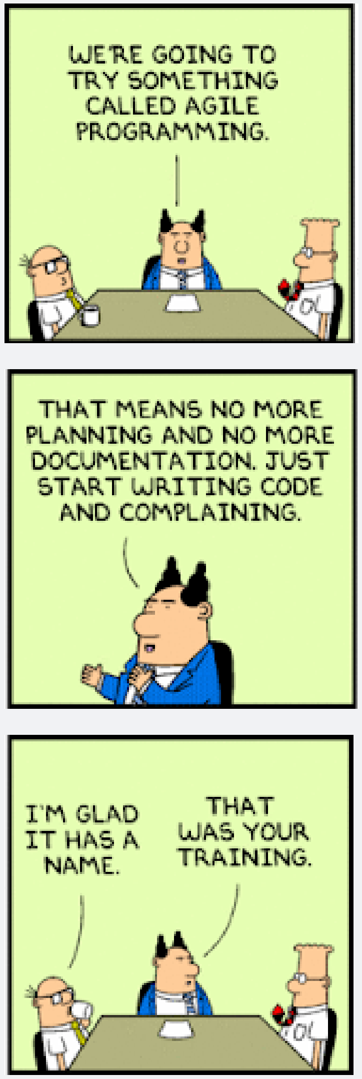
Where Agile Falls Short

The problem is that many software development projects still come in late and over-budget—there's room for improvement in Agile development practices. Behavior-driven development isn't designed to replace Agile development, but rather, improve upon it by adding more tools. These tools are designed to eliminate waste in order to help reduce software development costs and expedite delivery timelines.

There are several areas where traditional Agile development falls short:

- Transient User Stories: User stories are focused on user personas rather than business goals, and there's little ownership in keeping them up-to-date as the project evolves.

- Ad-hoc Acceptance Criteria: Acceptance criteria—or rules that let managers know when a user story is functionally complete—varies in quality. Often times, there's not enough information to create reliable tests for use in test-driven development.


WE'RE GOING TO TRY SOMETHING CALLED AGILE PROGRAMMING.
THAT MEANS NO MORE PLANNING AND NO MORE DOCUMENTATION. JUST START WRITING CODE AND COMPLAINING.
I'M GLAD IT HAS A NAME.
THAT WAS YOUR TRAINING.
©2007 Scott Adams, Inc.

- Limited Discovery: There's no common language used between business and technical teams. Without communicating well early on, it's easy for developers to go down a different path than business analysts imagined.

No Overarching Test: Test-driven development can ensure that the technical components of an application function as expected, but there's nothing that ties all of these components together to ensure that the software delivers real business value.

BDD places a heavy emphasis on concrete examples to reduce ambiguity in user stories and acceptance criteria. In addition to getting on the same page, these examples help uncover gaps that nobody had considered before the functionality is implemented. The feature files generated in the BDD process also serve as living documentation for the application, showing how it's supposed to function in a human-readable fashion.

Why You Should Use BDD

Behavior-driven development takes vague user stories and acceptance criteria and transforms them into a formal set of features and examples that can be used to generate documentation, automated tests, and a living specification. In other words, it gets everyone on the same page and ensures that there's no miscommunication about how the software behaves or what value it provides to the business.

At the very least, BDD is worth a try for almost any software project that requires input from stakeholders and business people. It's a great way to dramatically cut down on the waste that's typically seen in software projects while ensuring that they're delivered on time and on budget rather than becoming a statistic. The tests that you write will also be a lot more understandable and meaningful to everyone on the team.

Deliberate Discovery

Imagine a long-term software project that your team recently completed. How long did it take from inception to delivery? Now, imagine that you could do the same project over with everything kept the same—except your team would know everything they learned during the original project. How long would the second project take to complete? The difference between these two scenarios tells you that learning is the constraint in software development.

Deliberate discovery is the first step in behavior-driven development: Learning as quickly as possible to remove the constraints on a project to deliver on time and on budget.

1. What is Deliberate Discovery?

Most software development teams are familiar with user stories and acceptance criteria. For example, a user story for Twitter may state that a user can reply to another user's tweet. Acceptance criteria help define the specifics of how that functionality will be implemented, such as the presence of a reply button on the first user's profile. The problem is that these two tools—user stories and acceptance criteria—don't explore any unknowns.

Deliberate discovery involves having conversations about these user stories and acceptance criteria using concrete examples—and assuming ignorance. For example, a user might ask: Will my reply appear in my Twitter feed for anyone to see? The original user story and acceptance criteria may not have specified the answer to that question, but it would clearly have a big impact on the architecture of the overall application.

Rather than building software and putting it in front of users to get feedback, the goal of deliberate discovery is to try and learn as much as possible before writing any code to minimize waste and maximize productivity.

2. Who Should Be Involved?

Deliberate discovery processes should involve as many different team members as you need to provide insights into their specific areas of expertise. Developers may think of features on a very technical level, whereas domain experts may have insights into what actual customers are looking for in terms of functionality. All of these insights are critical for reducing the uncertainty around features and ultimately meeting the software's business goals.

Some examples of team members to include are:

- Product owners
- Business analysts
- Domain experts
- Users
- Developers
- UX designers
- Testers
- Ops engineers

3. Getting in the Right Mindset

Team exercises are a great way to get everyone in the right mindset for future deliberate discovery meetings. Liz Keogh suggests one popular exercise that works best in small groups of three or four people in a dedicated meeting room with a whiteboard.

The exercise starts by drawing four columns on a whiteboard:

- **Story Column**: Each person should tell a story about a problem they have encountered and a discovery they made to resolve the problem.
- **Commitment Column**: What decisions were made that solidified the problem? For example, decisions about deadlines or writing the wrong code.
- **Deliberate Discovery**: Could you have discovered information earlier that would have led to a different decision? For example, talking to customers or releasing early.
- **Real Options Column**: How could you have kept your options open longer? For example, making a commitment later when more information was available.

After finishing the exercise, the team should discuss how adding the discovery process could help them identify and avoid the problems. The takeaway should be that making the discovery early often prevents the problem from happening in the first place.

4. Running a Discovery Workshop

Discovery Workshops — also known as Three Amigos Meetings or Specification Workshops — are the cornerstone of the deliberate discovery process. The meetings should include any relevant non-technical stakeholders, developers, and testers. The goal of these meetings is to generate real-world examples of how the software should work if it existed, which can be high-level examples or actual executable specifications.

Discovery workshops should be kept short and frequent—usually about 25 minutes per user story—and no longer than one hour in total. When just starting out, it's a good idea to start with quick ad-hoc discovery workshops focused on the riskiest user stories, rather than trying to define concrete examples for all of the user stories at the onset. Once the team has a feel for it, the process can be expanded to include all user stories.

Example Mapping

The deliberate discovery process makes sense on paper—bringing together domain experts and developers to discuss the best path forward—but we haven't covered how to actually run these meetings. At the core, the most important outcomes of these meetings are concrete examples that can be turned into executable specifications, and ultimately, automated tests and living documentation for the software application.

Example mapping is one of the most popular strategies to come up with these concrete examples during deliberate discovery meetings. In addition to coming up with examples, the process helps uncover any unknowns or uncertain assumptions.

How Example Mapping Works

Example mapping was developed by Matt Wynne after he noticed that many deliberate discovery meetings were long, boring, and unstructured. His goal was to introduce a simple, low-tech method that could make these meetings short and productive. The example mapping process includes four components:

1. **User Story**: These are the user stories that were developed earlier in the deliberate discovery process and serve as the starting point for discussion.
2. **Rules**: The acceptance criteria for the user story that contain agreed-upon constraints about the scope of the story, and summarize the underlying examples.
3. **Examples**: These are the concrete examples covering each rule. There can be more than one example per rule to cover different potential use cases.
4. **Questions**: These are unknowns that arise when exploring the rules and examples or assumptions that were made in the interest of moving forward.

Matt suggests using a pack of four-color index cards and some pens to capture these different types of insights as the conversation unfolds, and then arranging these index cards in a map.

The low-tech approach is designed to eliminate any distractions and keep the team focused on having a meaningful conversation.

Running a Session

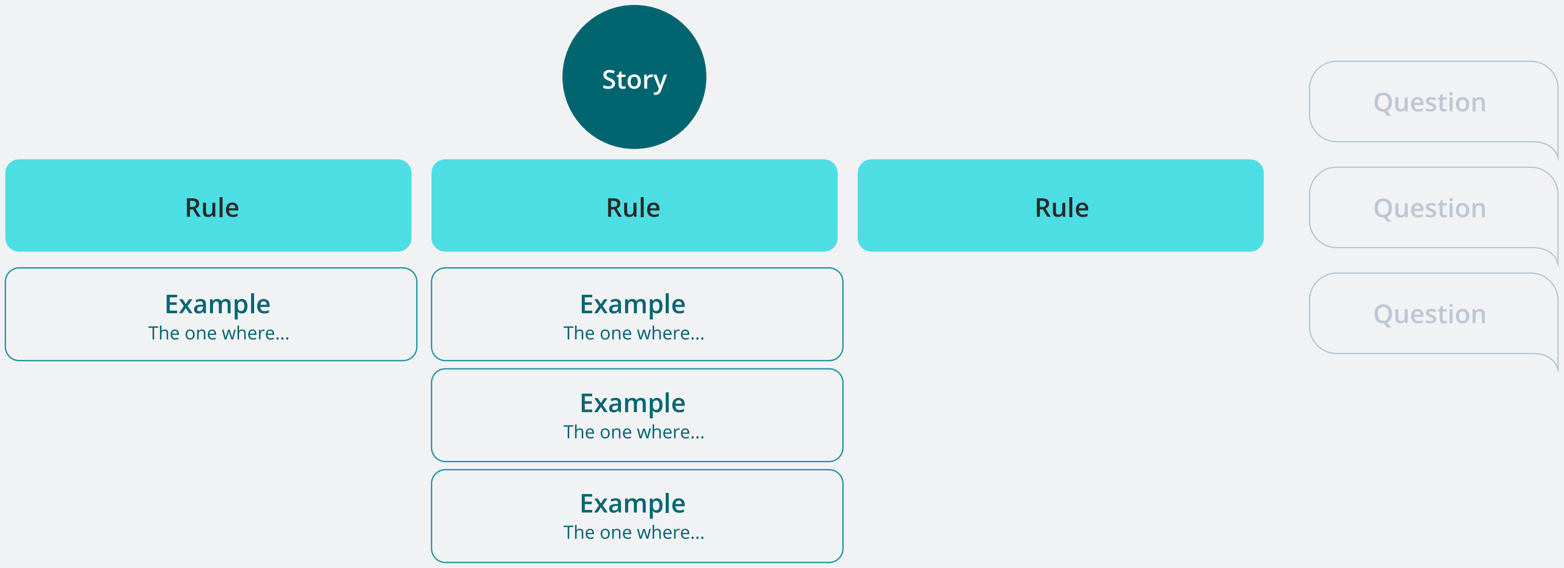
Example mapping sessions are most effective when they're timeboxed at about 25 minutes per user story. That way, the team has enough time to have a conversation and make an informed decision without spending all afternoon on theoretical concerns. The actual example mapping session involves four simple steps:

1. Write the story under discussion on a yellow card and place it on the table.
2. Write each acceptance criteria, or rule, on a blue card and put them across the table under the yellow card.
3. Come up with one or more concrete examples, write them on green cards, and place them under the appropriate rule.
4. If any questions come up, capture those on red cards and move on with the conversation.

After the time runs out, take a quick vote to see if the user story is ready to be pulled into development.

If too many blue cards on the table, the story may be too complex and could benefit from being sliced into multiple user stories. If there are too many red cards, the user story may be too complex and the product manager may have to refine the story.

It's important to keep in mind that a little uncertainty is normal in software development, so the team should try not to hold up the development process over minor disagreements or issues.

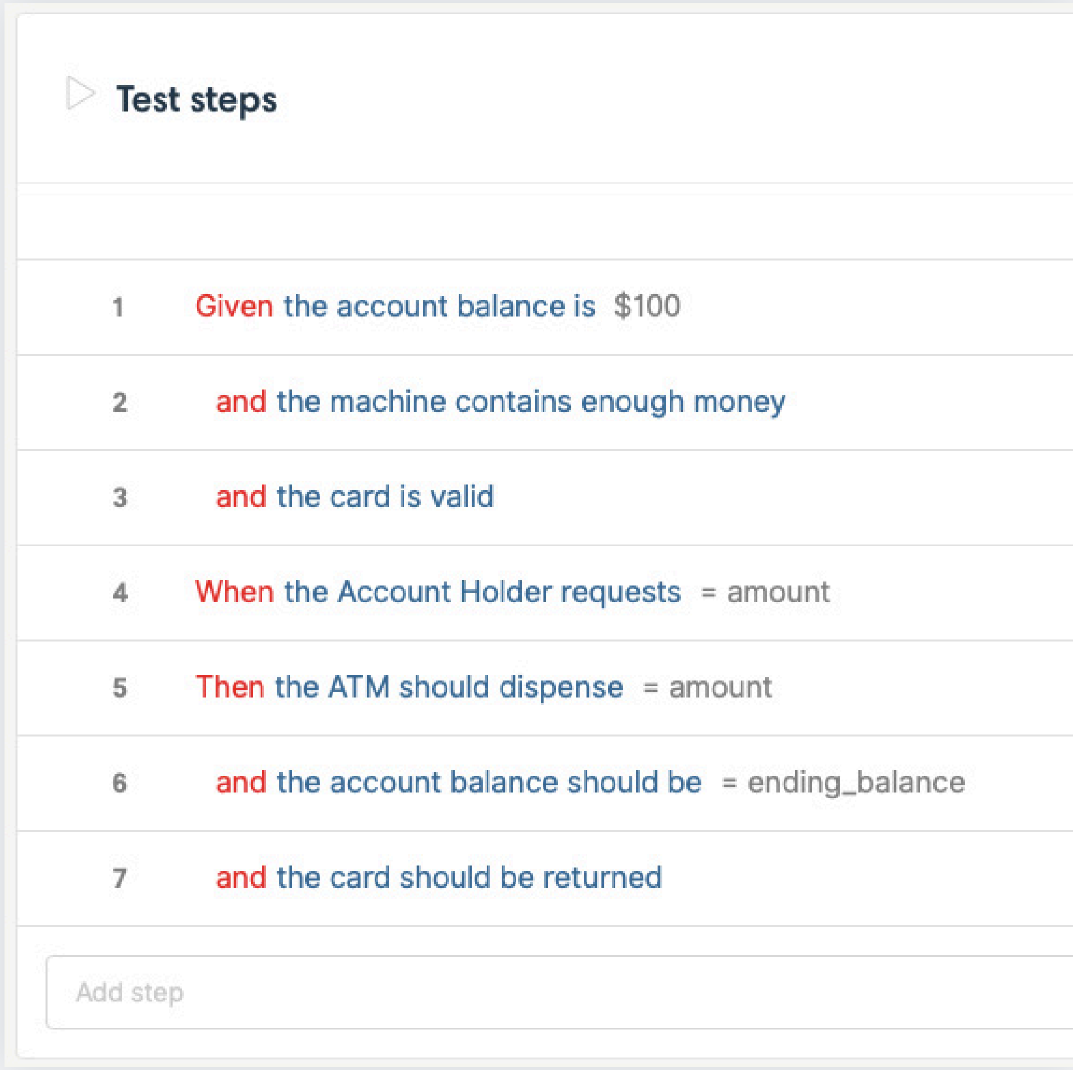


Recording the Results

Many development teams assume that the example mapping session should culminate in executable specifications (such as Cucumber scenarios). While this can be valuable, it can distract from the true purpose of the conversation. The idea is to reach a shared understanding of what it takes to get the story done rather than sitting in an IDE writing test in Gherkin. Fortunately, there's a middle ground that ensures everyone is on the same page.

HipTest improves the example mapping process by introducing a common interface for business and technical teams. Using the intuitive user interface, the team can create an executable specification without code, which can be automatically converted into a feature file. The process ensures that everyone is on the same page during the meeting and eliminates the work associated with manually writing the feature file in code.

In addition to entering these scenarios, HipTest helps you define and save common action words that can be used in many scenarios. This can further cut down on the time it takes to write a scenario during the example mapping session and get well thought out stories into production sooner.



Test steps

```
1  Given the account balance is  $100
2    and the machine contains enough money
3    and the card is valid
4  When the Account Holder requests  = amount
5  Then the ATM should dispense  = amount
6    and the account balance should be  = ending_balance
7    and the card should be returned
```

HipTest Scenarios

Building Executable Tests

The behavior-driven development process discussed thus far culminates in feature files that contain human-readable specifications. The next step is converting each of these specifications into executable tests using browser automation tools that simulate how users would actually be using the software product. These tests can then be added to the application's overall test coverage and any continuous integration strategies.

Let's take a look at how to convert feature files into executable specifications that actually verify that code meets the business requirements.

Writing Step Definitions

Most behavior-driven development frameworks automatically convert feature files into a step definition template. After this process runs, developers or test engineers must fill in the blanks to describe how the actual application should respond to the user actions.

For example, let's take a look at a simple Cucumber feature step definition using the Selenium WebDriver:

```
Scenario: Searching projects
Given I am on the search page
When I search for “Acme Project”
Then the page title should start
with “Acme Project”
```

The Gherkin file above becomes the following executable specification:

```
Given(/^I am on the search
page$/) dodriver = Selenium::Web-
Driver.for :chromedriver.get
“http://localhost:3000/search”end
```

```
When(/^I search for “([^”]*)”$/)
doelement = driver.find_element
(name: “q”)element.send_keys
“Acme Project”element.submitend
```

```
Then(/^the page title should
start with “([^”]*)”$/) dowait
= Selenium::WebDriver::Wait.new
(timeout: 10)wait.until { driver.
title.downcase.start_with? “acme”
}puts “Page title is #{driver.
title}”browser.closeend
```

There are many different browser automation tools that can be used in the process, and the right decision depends on your specific project. For example, Selenium remains the most popular overall framework, but many Ruby on Rails projects use Capybara as the preferred browser automation tool for use with Cucumber.

Automating the Process

There are many tools that automate the process of building step definitions. These tools can speed up the development process by automating work for test engineers, while ensuring that all step definitions are consistent in their design and usage.

HipTest Publisher Step Definition Export

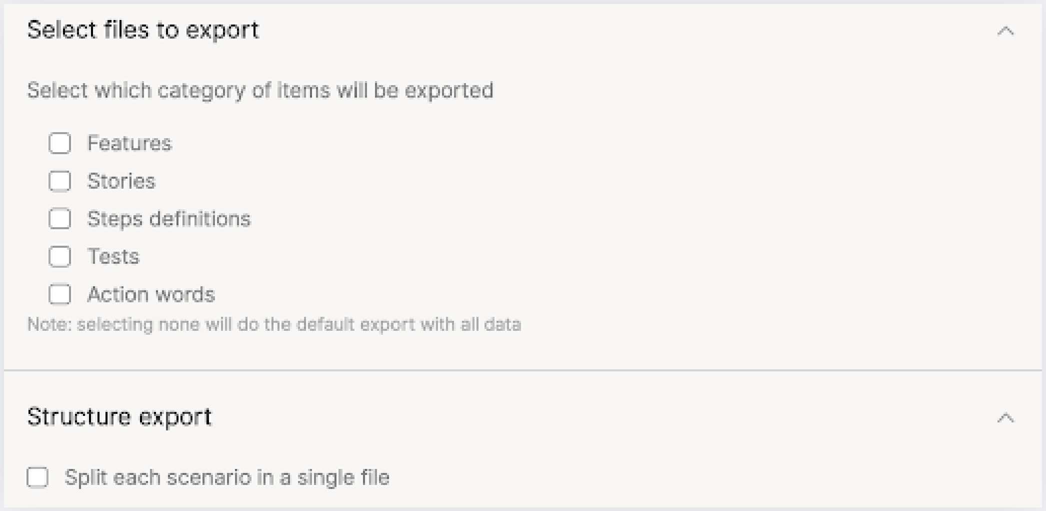
The HipTest Publisher tool transforms scenarios into executable tests in your chosen language and browser automation framework with the largest selection of automation frameworks and tools of any BDD platform. By defining individual steps describing the actions that are performed, you can avoid the need to go in and write any test code — or at least reduce the amount of test code that you need to write.

For example, you could define “Click on Profile” by setting a specific target, such as “a[text=”Profile]”, that will look for any link with the text “Profile” on the page. These steps can also include variables, although the use of variables means that test engineers may need to provide some input before the test can automatically execute

HipTest integrations can be either loose or tight:

- **Loose integration** avoids integrating automation-related content inside of scenarios and action words. The specific implementation is left up to the automation team to complete for the test code to run.
- **Tight integration** defines specific class names and other details to make the tests run out-of-the-box. That way, the testing can be completely automated.

HipTest Publisher Step Definition Export


Select files to export

Select which category of items will be exported

- [ ] Features
- [ ] Stories
- [ ] Steps definitions
- [ ] Tests
- [ ] Action words

Note: selecting none will do the default export with all data

Structure export

- [ ] Split each scenario in a single file

Most automated testing uses a combination of manual and automated processes. For example, it might not make sense to break down a very long set of instructions that will never be repeated into action words—it's easier to just write the test code by hand.

Important Considerations

It's important to watch out for blind spots when converting higher level feature files into lower level executable step definitions.

For example, suppose that you want to test a user's ability to update their profile. There may be several different ways that the user can save their profile information. Just because one button on the page works doesn't mean that the other methods won't be successful. Full test coverage may involve testing each method separately to ensure that there are no unexpected errors that occur.

You may also need to test scenarios in multiple browsers. For example, many businesses still have users on Internet Explorer. The latest version of Chrome won't necessarily function the same as an older version of Internet Explorer. Many browser automation frameworks enable you to switch between browsers to easily run tests covering multiple platforms.

Continuous Integration

Behavior-driven development (BDD) helps business and technical teams better communicate to create the right set of features, but the overarching goal of Agile software development is to deliver quality as fast and cheap as possible. The best way to accomplish that goal is through continuous integration, deployment, and delivery, which automatically ensures that any new code passes the appropriate tests before reaching production.

Let's take a closer look at continuous integration, how it fits into development workflows, and how to better integrate BDD into the process.

What is Continuous Integration?

Continuous integration is the practice of team members integrating their work frequently into a shared repository—often multiple times per day. Each integration is verified by an automated build to detect conflicts as quickly as possible before they reach a production environment. For example, a developer may commit new code to a git branch that's automatically checked by a continuous integration server against the master branch before it can merge.

There are several benefits to continuous integration:

- **Reduced Risk**: Defects can be immediately identified and fixed, which reduces the likelihood of a defect reaching a production environment.
- **More Confidence**: The reduced risk means that stakeholders can be confident in deploying new features without worrying about costly regressions.
- **Greater Visibility**: Everyone can see the project state in the shared code repository, which enables a better feedback cycle.

Behavior-driven development improves continuous integration by ensuring that effective tests can be run by the continuous integration service. Without effective tests, the CI server might let errors and omissions slip past into the production environment.

How CI Fits into Your Workflow

Continuous integration is the final step of most Agile development workflows. After writing tests and code, developers integrate their new additions into a shared repository and receive instant feedback about any conflicts or issues that need to be resolved.

A typical CI workflow looks something like this:

- Code and tests are committed to the repository.
- The software is built using a continuous integration service.
- The service runs all of the automated tests on the build.
- If the build fails, notifications are sent out to the team.
- If the build succeeds, the software is automatically deployed.

The workflow can be customized depending on the needs of the organization. For example, some teams prefer to deploy automated builds to a staging server rather than a production server, where final quality assurance can be done by hand. Other teams have CI servers that test new git branches against the master branch before it can be merged, and then automatically deploy any changes to the master branch.

Automating the Process

There are many different continuous integration servers that can automate the process, including both hosted or on-premise solutions. When choosing between these options, it's important to consider the cost and features. Open source on-premise solutions are free, but could require a lot of effort to configure and run, while hosted solutions can be more expensive, they are more convenient when it comes to customizations.

Some of the most popular CI solutions include:

- **CircleCI**: A leading CI platform that includes hosted and on-premise options with support for mobile platforms, including iOS and Android.
- **Jenkins**: The leading open source automation server with hundreds of different plugins to support building, deploying, and automating any project.
- **CodeShip**: A leading hosted CI platform that works well with a wide range of source code repositories and hosting environments.
- **TravisCI**: A popular solution that's commonly used with GitHub and open source repositories.

HipTest provides business and technical teams with greater visibility into the continuous integration process by automatically integrating with many popular continuous integration servers to make BDD processes more visible across the board.

There are a few easy steps in the process:

- Create a test run dedicated to the CI that includes only tests that have been fully automated.
- Install the HipTest Publisher tool on the CI tool and configure it to run using the step-by-step guide.
- Publish the results back to HipTest using the step-by-step guide to ensure that everyone is on the same page.

Living Documentation

Behavior-driven development does more than improve communication and reduce errors in production—it provides an up-to-date record of an application's features. In other words, it produces living documentation that's readable by both business and technical teams. This living documentation can be invaluable for many parts of the organization since it shows exactly what the application is supposed to do and proves that it can do it.

Let's take a look at how to leverage the scenarios and executable specifications created during the BDD processes discussed earlier to create a living documentation.

Why Documentation Matters

Documentation is instrumental in keeping stakeholders, developers, and users on the same page. While documentation may seem to counter Agile development principles by creating more work upfront, the process actually reduces the total work by ensuring that everyone is working together more effectively. There's no time wasted from miscommunications and it's much easier for everyone to get up-to-speed.

Some key benefits of documentation include:

- **Faster Update**s: Developers may write documentation for other developers making it easier for them to make updates. For example, system documentation makes it easy to find classes, methods, and variables in a large code base.
- **Improved Support**: Technical writers may create documentation for users that makes it easier for them to use the application, or help troubleshoot common problems. For example, Stripe's legendary API documentation was critical to its success.
- **Better Workflow**: Operational documentation helps stakeholders communicate their vision to development teams. For example, user stories, acceptance criteria, design documentation, and other resources provide context to the code.

The amount of documentation required for a project is where there's always some confusion. Agile development focuses on writing as little documentation as possible early on, while traditional waterfall development writes all of the documentation at the onset.

Where BDD Fits into the Picture

The behavior-driven development processes discussed in this article produce both human-readable scenarios expressed in Gherkin and executable specifications expressed in step definitions. When combined, these BDD tests communicate how features should function and ensure that the code properly executes those functions. Testing automation and continuous integration ensure that the features remain up-to-date over time.

Human-readable scenarios can be repurposed into system documentation since they effectively describe user stories and acceptance criteria. If a business analyst needs to know if a feature exists, they should be able to look at scenarios and see whether the step definitions have passed. If a developer wants to know the status of a feature, they should be able to run a test and see whether it has passed.

Building Living Documentation

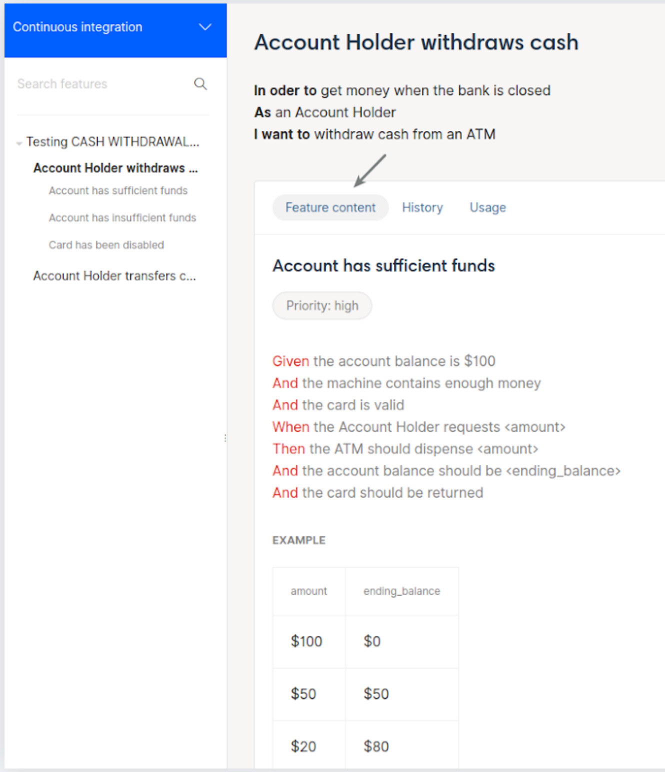
HipTest automatically generates up-to-date documentation when integrated with a continuous integration platform. Every time a BDD scenario is added and successfully run, the scenario is automatically added to the human-readable living documentation. This documentation can be easily accessed within HipTest by both business and technical teams without the need to open feature files and sort through a code repository.

If a feature fails or is removed, the living documentation is automatically updated to remove the feature without any effort by the business or technical teams. This helps ensure that everyone is always on the same page when it comes to an application's features and the status of each feature in the development process.

Where to Go From Here

Behavior-driven development is a game-changer for many Agile development teams. By dramatically improving communication, these processes minimize mistakes, reduce errors, and ensure a higher quality product reaches users.

HipTest makes it easy to implement and automate BDD processes with minimal coding, making it accessible to both business and technical teams. With its easy-to-use interface, it's the perfect solution for teams just starting out with BDD processes. The many features and integrations also make it perfect for established teams looking for a more streamlined approach to implementing BDD across their organization.


Continuous integration
Search features
Testing CASH WITHDRAWAL...
Account Holder withdraws ...
Account has sufficient funds
Account has insufficient funds
Card has been disabled
Account Holder transfers c...
Feature content
History
Usage
Priority: high

Account Holder withdraws cash

In oder to get money when the bank is closed
As an Account Holder
I want to withdraw cash from an ATM

Account has sufficient funds

```
Given the account balance is $100
And the machine contains enough money
And the card is valid
When the Account Holder requests <amount>
Then the ATM should dispense <amount>
And the account balance should be <ending_balance>
And the card should be returned
```

EXAMPLE

amount	ending_balance
$100	$0
$50	$50
$20	$80

About HipTest

HipTest is the only BDD collaboration platform for defining acceptance criteria, executing automated tests, and generating living documentation.

Here's how HipTest helps you deliver quality at the speed of modern business:

| Put collaboration at the forefront

| Accelerate development with automated testing

| Get real-time insights with living documentation

SMARTBEAR HipTest

Start Your Free Trial

SMARTBEAR