JavaScript is not only widely used to create attractive user interfaces for the Web, but with the advent of Node.js, it is also becoming a very popular and powerful language to write server-side applications. Using behavior-driven development and some common testing patterns and best practices, you will be able to avoid these traps.

This book will show you how to do BDD in a practical way. We will start with a basic introduction of what BDD is and why the classical approach to testing has failed. Afterwards, we will dive directly into an introduction to Node.js, Mocha, and Sinon.JS. Finally, we will cover more advanced subjects such as how to write a fast and effective test suite for a RESTful web API, and how to do the same with a rich UI using Cucumber.js and Protractor.

This book will give you great insight into how to architect your system to make it more testable and modular, but at the same time avoid the nightmare of abusing mock objects.

Who this book is written for
This book is ideal for any JavaScript developer who is interested in producing well-tested code. If you have no prior experience with testing, Node.js, or any other tool, do not worry, as they will be explained from scratch.

What you will learn from this book
- Understand the basic concepts of BDD and how it is different from classic unit testing
- Divide your system into different modules that can be tested separately, but at the same time not falling into the trap of unit testing
- Use Mocha, Sinon.JS, and Chai to write expressive BDD features
- Implement Cucumber.js to automate tests written in Gherkin so that your stakeholders can understand them
- Discover how to test asynchronous systems, either based on callbacks or promises
- Test a RESTful web API and a rich UI using WebDriverJS and Protractor
- Refactor and keep your test code base maintainable using best practices and patterns such as PageObject
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Welcome to BDD'
- A synopsis of the book’s content
- More information on *Learning Behavior-driven Development with JavaScript*

**About the Author**

Enrique Amodeo is an experienced software engineer currently working and living in Berlin. He is a very eclectic professional with very different interests and more than 15 years of experience. Some of his areas of expertise are JS, BDD/TDD, REST, NoSQL, object-oriented programming, and functional programming.

As an agile practitioner, he uses BDD and emergent design in his everyday work and tries to travel light. Experienced in both frontend and server-side development, he has worked with several technical stacks, including Java/JEE, but since 2005, he prefers to focus on JS and HTML5. He is now very happy to be able to apply his JS knowledge to the server-side development, thanks to projects such as Node.js.

He also has written a book in Spanish on designing web APIs, following the REST and hypermedia approach (https://leanpub.com/introduccion apis rest). I would like to thank my wife for making this book possible. She is the one who supported me and reminded me to "continue writing that difficult chapter" whenever I started thinking of doing something else. Without her, I would probably have never completed this book!
Learning Behavior-driven Development with JavaScript

JavaScript is not only widely used to create attractive user interfaces for the Web, but, with the advent of Node.js, it is also becoming a very popular and powerful language with which to write server-side applications. In this context, JavaScript systems are no longer toy applications, and their complexity has grown exponentially. To create complex applications that behave correctly, it is almost mandatory to cover these systems with an automated test suite. This is especially true in JavaScript because it does not have a compiler to help developers. Unfortunately, it is easy to fall into testing pitfalls that will make your test suite brittle; hard to maintain, and sooner or later, they will become another headache instead of a solution. Using behavior-driven development and some common testing patterns and best practices, you will be able to avoid these traps.

A lot of people see the whole TDD/BDD approach as a black-and-white decision. Either you do not do it, or you try to achieve a hundred percent test coverage. The real world calls for a more pragmatic approach: write the tests that really pay off and do not write those that do not give you much value. To be able to take this kind of decision, a good knowledge of BDD and the costs associated with it is needed.

What This Book Covers

Chapter 1, Welcome to BDD, presents the basic concepts that act as a foundation for BDD. Its goal is to debunk a few false preconceptions about BDD and to clarify its nomenclature. It is the only theoretical chapter in the whole book.

Chapter 2, Automating Tests with Mocha, Chai, and Sinon, introduces the basic tools for testing in JavaScript. We will go through the installation process and some simple examples of testing. You can safely skip this chapter if you are well versed with these tools.

Chapter 3, Writing BDD Features, presents some techniques for transforming a functional requirement written in normal language into a set of automated BDD tests or features. We will write our first BDD feature.

Chapter 4, Cucumber.js and Gherkin, repeats the exercise of the previous chapter but this time using Cucumber.js. This way we can compare it fairly with Mocha. You can safely skip this chapter if you already know Cucumber.js.

Chapter 5, Testing a REST Web API, shows you how to test not only the core logic, but also the Node.js server that publishes a Web API. This chapter will be of special interest if you are writing a REST API.
Chapter 6, *Testing a UI Using WebDriverJS*, shows you how to approach testing the UI layer from the perspective of BDD. You will also learn about WebDriverJS and how it can help you in this task.

Chapter 7, *The Page Object Pattern*, explains how to create robust UI tests that are less susceptible to being broken by UI design changes. For that, we will apply the page object pattern.

Chapter 8, *Testing in Several Browsers with Protractor and WebDriver*, shows you how to use the Protractor framework to run your test suite in several browsers.

Chapter 9, *Testing Against External Systems*, gives you some basic techniques for doing this and, most important, shows you when not to do it. Although this kind of test is not strictly BDD, sometimes you do need to test against external systems.

Chapter 10, *Final Thoughts*, briefly summarizes the book and clarifies the right granularity for BDD testing. It will also tell you whether to do only BDD at the core business level, or add additional tests at other levels.
Welcome to BDD

Before we start coding tests, we need to understand what behavior-driven development (BDD) is and how it differs from test-driven development (TDD).

We need to understand not only the concept of BDD, but also all the jargon associated with it. For example, what is a feature? Or what is a unit test? So, in this chapter, I will try to clarify some common vocabulary in order to give you a solid understanding of what every technical term means.

In this chapter, you will learn:

- The reason for writing automated tests
- The workflow prescribed by the test-first approach
- What BDD is and how it differs from TDD
- What a unit test really is
- The different phases that compose a test
- What test doubles are and the different kinds of test doubles that exist
- The characteristics of a good test

The test-first approach

Testing is nothing new in software engineering; in fact, it is a practice that has been implemented right from the inception of the software industry, and I am not talking only about manual testing, but about automated testing as well. The practice of having a set of automated tests is not exclusive to TDD and BDD, but it is quite old. What really sets apart approaches such as TDD and BDD is the fact that they are test-first approaches.
In traditional testing, you write your automated test after the code has been written. At first sight, this practice seems to be common sense. After all, the point of testing is discovering bugs in the code you write, right? Probably, these tests are executed by a different team than the one that wrote the code in order to prevent the development team from cheating.

Behind this traditional approach lies the following assumptions:

- Automated tests can discover new bugs
- The project is managed under a waterfall life cycle or similar, where large chunks of functionality are developed until they are perfect, and only then is the code deployed

These assumptions are mostly false nowadays. Automated tests cannot discover anything new but only provide feedback about whether the code behaves as specified or expected. There can be errors in the specification, misunderstandings, or simply different expectations of what is correct between different people. From the point of view of preventing bugs, automated tests are only good as regression test suites. A regression test suite contains tests that prove that a bug that is already known is fixed. Since there usually exists a lot of misunderstanding between the stakeholders themselves and the development team, the actual discovery of most bugs is often done during exploratory testing or by actual users during the beta or alpha phase of the product.

About the waterfall approach, the industry has been moving away from it for some time now. It is clearly understood that not only fast time to market is crucial, but that a project’s target can undergo several changes during the development phase. So, the requirements cannot be specified and set in stone at the beginning of the project. To solve these problems, the agile methodologies appeared, and now, they are starting to be widely applied.

Agile methodologies are all about fast feedback loops: plan a small slice of the product, implement it, and deploy and check whether everything is as expected. If everything is correct, at least we would already have some functionality in production, so we could start getting some form of benefit from it and learn how the user engages with the product. If there is an error or misunderstanding, we could learn from it and do it better in the next cycle. The smaller the slice of the product we implement, the faster we will iterate throughout the cycles and the faster we will learn and adapt to changes. So ideally, it is better to build the product in small increments to be able to obtain the best from these feedback loops.
This way of building software changed the game, and now, the development team needs to be able to deliver software with a fast pace and in an incremental way. So, any good engineering practice should be able to enable the team to change an existing code base quickly, no matter how big it is, without a detailed full plan of the project.

The test-first cycle

In this context, the test-first approach performs much better than the traditional one. To understand why, first, let’s have a look at the test-first cycle:

As you can see, the cycle starts with a new coding task that represents any sensible reason to change the codebase. For example, a new functionality or a change in an existing one can generate a new coding task, but it can also be triggered by a bug. We will talk a bit more in the next section about when a new coding task should trigger a new test-first cycle.
Write a failing test
Once we have a coding task, we can engage in a test-first cycle. In the first box of the previous diagram, write a failing test, we try to figure out which one is the simplest test that can fail; then, we write it and finally see it fail.

Do not try to write a complex test; just have patience and go in small incremental steps. After all, the goal is to write the simplest test. For this, it is often useful to think of the simplest input to your system that will not behave as expected. You will often be surprised about how a small set of simple tests can define your system!

Although we will see this in more detail in the upcoming chapters, let me introduce a small example. Suppose we are writing the validation logic of a form input that takes an e-mail and returns an array of error messages. According to the test-first cycle, we should start writing the most simple test that could fail, and we still have not written any production code. My first test will be the success case; we will pass a valid e-mail and expect the validation function to return an empty array. This is simple because it establishes an example of what is valid input, and the input and expectations are simple enough.

Make the test pass
Once you have a failing test, you are allowed to write some production code to fix it. The point of all of this is that you should not write new code if there is not a good reason to do so. In test-first, we use failing tests as a guide to know whether there is need for new code or not. The rule is easy: you should only write code to fix a failing test or to write a new failing test.

So, the next activity in the diagram, make the test pass, means simply to write the required code to make the test pass. The idea here is that you just write the code as fast as you can, making minimal changes needed to make the test pass. You should not try to write a nice algorithm or very clean code to solve the whole problem. This will come later. You should only try to fix the test, even if the code you end up writing seems a bit silly. When you are done, run all the tests again. Maybe the test is not yet fixed as you expected, or your changes have broken another test.

In the example of e-mail validation, a simple return statement with a empty array literal will make the test pass.
Clean the code

When all the tests are passing, you can perform the next activity, **clean the code**. In this activity, you just stop and think whether your code is good enough or whether it needs to be cleaned or redesigned. Whenever you change the code, you need to run all the tests again to check that they are all passing and you have not broken anything. Do not forget that you need to clean your test code too; after all, you are going to make a lot of changes in your test code, so it should be clean.

How do we know whether our code needs some cleaning? Most developers use their intuition, but I recommend that you use good, established design principles to decide whether your code is good enough or not. There are a lot of established design principles around, such as the SOLID principles (see [http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf](http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf)) or Craig Larman's GRASP patterns (see [http://www.craигlarman.com/wiki/index.php?title=Books_by_Craig_Larman#Applying_UML_and_Patterns](http://www.craiglarman.com/wiki/index.php?title=Books_by_Craig_Larman#Applying_UML_and_Patterns)). Unfortunately, none of the code samples of these books are in JavaScript, so I will summarize the main ideas behind these principles here:

- **Your code must be readable.** This means that your teammates or any software engineer who will read your code 3 months later should be able to understand the intent of the code and how it works. This involves techniques such as good naming, avoiding deep-nested control structures, and so on.
- **Avoid duplication.** If you have duplicated code, you should refactor it to a common method, class, or package. This will avoid double maintenance whenever you need to change or fix the code.
- **Each code artifact should have a single responsibility.** Do not write a function or a class that tries to do too much. Keep your functions and objects small and focused on a single task.
- **Minimize dependencies between software components.** The less a component needs to know about others, the better. To do so, you can encapsulate internal state and implementation details and favor the designs that interchange less information between components.
- **Do not mix levels of abstractions in the same component; be consistent in the language and the kind of responsibility each component has.**
To clean your code, you should apply small refactoring steps. Refactoring consists of a code change that does not alter the functionality of the system, so the tests should always pass before and after each refactoring session. The topic of refactoring is very big and out of the scope of this book, but if you want to know more about it, I recommend Refactoring: Improving the Design of Existing Code (http://martinfowler.com/books/refactoring.html).

Anyway, developers often have a good instinct to make their code better, and this is normally just enough to perform the clean code step of the test-first cycle. Just remember to do this in small steps, and make sure that your tests pass before and after the refactoring.

In a real project, there will be times when you just do not have much time to clean your code, or simply, you know there is something wrong with it, but you cannot figure out how to clean it at that moment. In such occasions, just add a TODO comment in your code to mark it as technical debt, and leave it. You can talk about how to solve the technical debt later with the whole team, or perhaps, some iterations later, you will discover how to make it better.

Repeat!

When the code is good enough for you, then the cycle will end. It is time to start from the beginning again and write a new failing test. To make progress, we need to prove with a failing test whether our own code is broken!

In our example, the code is very simple, so we do not need to clean up anything. We can go back to writing a failing test. What is the most simple test that can make our code fail? In this case, I would say that the empty string is an invalid e-mail, and we expect to receive an email cannot be empty error. This is a very simple test because we are only checking for one kind of error, and the input is very simple; an empty string.

After passing this test, we can try to introduce more tests for other kinds of errors. I would suggest the following order, by complexity:

- Check for the presence of an @ symbol
- Check for the presence of a username (@mailcompany.com should fail, for example)
- Check for the presence of a domain (peter@ should fail too)
- Check whether the domain is correct (peter@bad#domain!com should fail)
After all of these tests, we would probably end up with a bunch of `if` statements in our code. It is time to refactor to remove them. We can use a regular expression or, even better, have an array or validation rules that we can run against the input.

Finally, after we have all the rules in place and our code looks clean, we can add a test to check for several errors at the same time, for example, checking that `@bad#domain!com` should return an array with the **missing username** and **incorrect domain** errors.

What if we cannot write a new failing test? Then, we are simply done with the coding task!

As a summary, the following are the five rules of the test-first approach:

- Don't write any new tests if there is not a new coding task.
- A new test must always fail.
- A new test should be as simple as possible.
- Write only the minimum necessary code to fix a failing test, and don't bother with quality during this activity.
- You can only clean or redesign your code if all the tests pass. Try to do it in each cycle if possible.

**Consequences of the test-first cycle**

This way of writing code looks weird at first and requires a lot of discipline from the engineers. Some people think that it really adds a big overhead to the costs of a project. Maybe this is true for small projects or prototypes, but in general, it is not true, especially for codebases that need to be maintained during periods of over 3 or 4 months.

Before test-first, most developers were doing manual testing anyway after each change they made to the code. This manual testing was normally very expensive to achieve, so test-first is just cutting costs by automating such activity and putting a lot of discipline in our workflow.

Apart from this, the following are some subtle consequences:

- Since you write tests first, the resulting code design ends up being easily testable. This is important since you want to add tests for new bugs and make sure that changes do not break the old functionality (regression).
Welcome to BDD

• The resulting codebase is minimal. The whole cycle is designed to make us write just the amount of code needed to implement the required functionality. The required functionality is represented by failing tests, and you cannot write new code without a failing test. This is good, because the smaller the code base is, the cheaper it is to maintain.

• The codebase can be enhanced using refactoring mechanisms. Without tests, it is very difficult to do this, since you cannot know whether the code change you have done has changed the functionality.

• Cleaning the code in each cycle makes the codebase more maintainable. It is much cheaper to change the code frequently and in small increments than to do it seldom and in a big-bang fashion. It is like tidying up your house; it is better to do it frequently than do it only when you expect guests.

• There is fast feedback for the developers. By just running the test suite, you know, in the moment, that the changes in the code are not breaking anything and that you are evolving the system in a good direction.

• Since there are tests covering the code, the developers feel more comfortable adding features to the code, fixing bugs, or exploring new designs.

There is, perhaps, a drawback: you cannot adopt the test-first approach easily in a project that is in the middle of its development and has been started without this approach in mind. Code written without a test-first approach is often very hard to test!

BDD versus TDD

The test-first approach covered in the previous section is what has been described generically as TDD.

The problem with TDD, as already presented, is that it does not say anything about what a coding task is, when a new one should be created, or what kind of changes we should allow.

It is clear that a change in a requisite or a newly discovered bug should trigger a TDD cycle and involve a new coding task. However, some practitioners think that it is also OK to change the codebase, because some engineer thought that a change in the technical design would be good for the system.

The biggest problem in classic TDD is that there is a disconnection between what the product is supposed to do and what the test suite that the development team builds is testing. TDD does not explicitly say how to connect both worlds. This leads to a lot of teams doing TDD, but testing the wrong things. Yes, perhaps they were able to test all their classes, but they tested whether the classes behave as expected, not whether the product behaves as expected.
Yes, perhaps they have a very detailed test suite with high coverage and with all its tests passing, but this offers no clue about whether the product itself will work as expected or whether a bug is resolved. This is a bad situation, as the main benefit of the tests is in the fast feedback they provide.

BDD tries to fix these problems by making the test suite directly dependent of the feature set of the product. Basically, BDD is a test-first approach where a new coding task can be created only when a change in the product happens: a new requisite, a change in an existing one, or a new bug.

This clarification changes rule 1 of test-first, from *Don't write any new tests if there is not a new coding task* to *Don't write any new tests if there is not a change in the product*. This has some important implications, as follows:

- You should not add a new class or function or change the design if there is not a change in the product. This is a more specific assertion about coding tasks than the generic one about TDD.
- As a change in the product always represents only a feature or bug, you only need to test features or bugs, not components or classes. There is no need to test individual classes or functions. Although this does not mean that it is a bad idea to do so, such tests are not viewed as essential from the BDD point of view.
- Tests are always about describing how the product behaves and never about technical details. This is a key difference with TDD.
- Tests should be described in a way that the stakeholders can understand to give feedback about whether they reflect their expected behavior of the system. That is why, in BDD jargon, tests are not called tests, but specifications or features.
- Test reports should be understandable for the stakeholders. This way, they can have direct feedback of the status of the project, instead of having the need for the chief architect to explain the test suite result to them.
- BDD is not only an engineering practice, but it needs the team to engage frequently with the stakeholders to build a common understanding of the features. If not, there would be a big risk that we are testing the wrong feature.

Of course, there were teams that practiced TDD in this way, avoiding all of the problems mentioned earlier. However, it was Dan North who first coined the term BDD to this specific way of doing TDD and to popularize this way of working.

BDD exposes a good insight: we should test features instead of components. This is very important from the perspective of how to design a good test suite. Let's explore this subject a bit in the next section.
Exploring unit testing

99.99 percent of the projects we are going to face will be complex and cannot be tested with a single test. Even small functionalities that a non-engineer would consider very simple will actually be more complex than expected and have several corner cases. This forces us to think about how to decompose our system in tests or, in other words, what exactly are the tests that we should write.

In the beginning of the test-first movement, there was no clear answer to this question. The only guidance was to write a test for each unit and make the tests from different units independent between them.

The notion of units is very generic and does not seem to be very useful to guide the practice of test-first. After a long debate in the community, it seems that there is a consensus that there exists at least two kinds of units: features and components.

A feature is a single concrete action that the user can perform on the system; this will change the state of the system and/or make the system perform actions on other third-party systems. Note that a feature is usually a small-grained piece of functionality of the system, and a use case or user story can map to several features. An important thing about features is that they describe the behavior of the system from the point of view of the user. Slicing a user story into features is a key activity of BDD, and throughout the book, we will see plenty of examples of how to do it.

The other kinds of units are the components of our system. A component is any software artifact, such as classes, procedures, or first-order functions, that we use to build the system.
We can conceptualize any product we are building as a matrix of features versus components, like in the following image:

In this image, we can see that any system implements a set of features, and it is implemented by a set of components. The interesting thing is that there is seldom a one-to-one relationship between components and features. A single feature involves several components, and a single component can be reused across several features.
With all this in mind, we can try to understand what traditional TDD, or traditional unit testing, is doing. In the traditional approach, the idea is to make unit tests of components. So, each component should have a test of its own. Let's have a look at how it works:

In the preceding image, you can see that the system is built incrementally, one component at a time. The idea is that with each increment, a new component is created or an existing one is upgraded in order to support the features. This has the advantage that if a test is failing, we know exactly which component is failing.
Although this approach works in theory, in practice, it has some problems. Since we are not using the features to guide our tests, they can only express the expected behavior of the components. This usually generates some important problems, such as the following ones:

- There is no clear and explicit correlation between the components and the features; in fact, this relationship can change over time whenever there is a design change. So, there is no clear progress feedback from the test suite.
- The test results only make sense for the engineering team, since it is all about components and not the behavior of the systems. If a component test is failing, which features are failing? Since there is not a clear correlation between features and components, it is expensive to answer this question.
- If there is a bug, we don't know which tests to modify. Probably, we will need to change several tests to expose a single bug.
- Usually, you will need to put a lot more effort into your technical design to have a plan of what components need to be built next and how they fit together.
- The tests are checking whether the component behaves according to the technical design, so if you change the design, then you need to change the tests. The whole test suite is vulnerable to changes in the design, making changes in the design harder. Hence, a needed refactor is usually skipped, making the whole quality of the codebase worse and worse as time passes.

Of course, a good and experienced engineering team can be successful with this approach, but it is difficult. It is not surprising that a lot of people are being very vocal against the test-first approach. Unit test components is the classic and de facto approach to test-first, so when someone says terms such as TDD or unit testing, they usually mean component unit testing. This is why problems with component unit testing have been wrongly confused with problems of the general test-first approach.
The other way of doing test-first is to unit test features, which is exactly what BDD make us do. We can have a look at the diagram to see how a system progresses using BDD:

As we can see, as time progresses, we add tests for each feature, so we can have good feedback about the status of completion of the project. If a test is failing, then it means that the corresponding feature is broken.

On the other hand, we don't need a very detailed up-front design to start coding. After all, we have the guidance of the behavior of the system to start the test-first workflow, and we can fine-tune our design incrementally using the "clean code" step of the workflow. We can discover components on the fly while we are delivering features to the customer. Only a high-level architecture, some common conventions, and the set of tools and libraries to use, are needed before starting the development phase. Furthermore, we can isolate the test from most technical changes and refactorings, so in the end, it will be better for our codebase quality.

Finally, it seems to be common sense to focus on the features; after all, this is what the customer is really paying us for. Features are the main thing we need to ensure that are working properly. An increment in component unit testing does not need to deliver any value, since it is only a new class, but an increment in BDD delivers value, since it is a feature. It does not matter whether it is a small feature or not; it is a tangible step forward in project termination.
There is, of course, a disadvantage in this approach. If a test is failing, we know which feature is failing, but we do not know which component needs to be fixed. This involves some debugging. This is not a problem for small and medium systems, since a feature is usually implemented by 3–5 components. However, in big systems, locating the affected component can be very costly.

There is no silver bullet. In my opinion, BDD is an absolute minimum, but unit testing some of the key components can be beneficial. The bigger the system is, the more component unit testing we should write, in addition to the BDD test suite.

The structure of a test

As we saw earlier, a unit could be a feature if we are doing BDD, or it could be a component if we are doing traditional TDD. So, what does a unit test look like? From a very high level point of view, a unit test is like the following image:

You can see that the test is acting as the unit. The term "act" means that the test is performing a single operation on the unit through its public API.
Then, the test must check or assert the result of the operation. In this phase, we need to check whether the actual return value is as we expect, but we also need to check whether the side effects are the expected ones. A side effect is a message that the unit sends to other units or third-party systems in order to perform the action correctly.

Side effects look quite abstract, but in fact, they are very simple. For example, from the point of view of traditional TDD, a side effect can be a simple call from one class to another. From the point of view of BDD, a side effect can be a call to a third-party system, such as an SMS service, or a write to the database.

The result of an action will depend on the prior state of the system we are testing. It is normal that the expected result of the very same action varies according to the specific state the system is in. So, in order to write a test, we need to first set up or arrange the system in a well-known state. This way, our test will be repeatable.

To sum up, every test must have the following three phases:

- **Set up/Arrange**: In this phase, we set up the state of the system in a well-known state. This implies choosing the correct input parameters, setting up the correct data in the database, or making the third-party systems return a well-known response.
- **Act**: In this phase, we perform the operation we are testing. As a general rule, the act phase of each test should involve only one action.
- **Assert**: In this phase, we check the return value of the operation and the side effects.

**Test doubles**

Whenever we see the term "unit testing", it means that we are making tests of the units of our system in an isolated way. By isolated, I mean that each test must check each unit in a way independent of the others. The idea is that if there is a problem with a unit, only the tests for that unit should be failing, not the others. In BDD, this means that a problem in a feature should only make the tests fail for that feature. In component unit testing, it means that a problem with a component (a class, for example) should only affect the tests for that class. That is why we prescribe that the act phase should involve only one action; this way, we do not mix behaviors.

However, in practice, this is not enough. Usually, features can be chained together to perform a user workflow, and components can depend on other components to implement a feature.
This is not the only problem, as we saw earlier; it is usually the case that a feature needs to talk with other systems. This implies that the set up phase must manipulate the state of these third-party systems. It is often unfeasible to do so, because these systems are not under our control. Furthermore, it can happen that these systems are not really stable or are shared by other systems apart from us.

In order to solve both the isolation problem and the set up problem, we can use test doubles. Test doubles are objects that impersonate the real third-party systems or components, just for the purpose of testing. There are mainly the following type of test doubles:

- **Fakes**: These are a simplified implementation of the system we are impersonating. They usually involve writing some simple logic. This logic should never be complex; if not, we will end up reimplementing such third-party systems.

- **Stubs**: These are objects that return a predefined value whenever one of its methods is called with a specific set of parameters. You can think of them as a set of hardcoded responses.

- **Spies**: These are objects that record their interactions with our unit. This way, we can ask them later what happened during our assertion phase.

- **Mocks**: These are self-validating spies that can be programmed during the set up phase with the expected interaction. If some interaction happens that is not expected, they would fail during the assertion phase.

We can use spies in the assertion phase of the test and stubs in the set up phase, so it is common that a test double is both a spy and a stub.

In this book, we will mostly use the first three types, but not mocks, so don't worry much about them. We will see plenty of examples for them in the rest of the book.

**What is a good test?**

When we are writing tests, we need to keep in mind a series of guidelines in order to end up with a useful test suite. Every test we write should have the following properties:

- They should be relevant. A test must be relevant from the point of view of the product. There is no point in testing something that, when it is done, does not clearly move the project forward to completion. This is automatically achieved by BDD, but not by traditional TDD.
Welcome to BDD

• They should be repeatable. Tests must always offer the same results if there has not been a code change. If it is failing, you must change the code to see it pass, and if it is passing, it must not start failing if nobody changed the code. This is achieved through a correct setup of the system and the use of test doubles. If tests are not repeatable, they offer no useful information! I have seen teams ignore tests that are flipping between passing and failing because of incorrect setup or race conditions. It would have been better not to waste time and money in writing a test that nobody cares about because it is not reliable.

• They should be fast. After all, one key point of test-first is rapid feedback and quick iteration. It is not very cost effective if you need to wait 15 minutes for the tests to end whenever you make a code change in a test-first cycle.

• They should be isolated. A test should fail only because the feature (or component) it is testing has a defect. This will help us diagnose the system to pinpoint where the error is. This will help us write code in an incremental fashion in the order our customers require (often, the most valuable features first). If the test is not isolated, then we often cannot write a new test, because we need first to write another feature or component that this one depends on.

Summary
The test-first approach appeared as an engineering practice to back up the agile methodologies. It supports the notion of incremental design and implementation of the codebase in order to be able to deliver software fast, incrementally, and in short iterations.

The test-first approach tells us to first write the most simple failing test that we can think of, fix it with the smallest change of code possible, and finally, clean the code, changing the design if necessary and taking advantage of the fact that we have tests as a safety net. Repeat the cycle until there is no new failing test to write.

There are two main approaches to test-first: traditional TDD and BDD. In traditional TDD, or component unit testing, we test components (classes, functions, and so on) in isolation from other components. In BDD, we test simple user actions on the system, also known as features, in isolation from other features. Both are forms of unit testing, but due to historic reasons, we reserve the term "unit testing" for component unit testing.
In my opinion, the BDD approach is superior, because it relates the tests with the actual behavior of the system, making the progress of the project more visible, focusing the team on what really matters and decoupling the tests themselves from the specific details of the technical design. However, in big systems, it can be difficult to diagnose which components should be fixed when a feature fails, so some degree of traditional TDD is still useful.

Tests should be isolated to avoid coupling between them and enable fast detection of which feature/component must be fixed. They should also be fast to get a quick feedback cycle during development. Furthermore, tests should be repeatable; if not, we cannot trust their result, and they become a waste of time and money.

To make tests isolated, fast, and repeatable, we can use test doubles. They replace and impersonate third-party systems or components in our test suite. They can be used both to set up the system in a predictable way, hence achieving repeatability and quick execution, and to check the side effects produced by the system under test. In traditional unit testing, we can use them to isolate the component under test from other components.

This concludes the first chapter. Fortunately, it is the only one devoted to theory in this book. In the next chapter we will start coding!
Where to buy this book

You can buy Learning Behavior-driven Development with JavaScript from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.