SAMPLE CHAPTER

BDD IN ACTION

Behavior-Driven Development for the whole software lifecycle

John Ferguson Smart

FOREWORD BY Dan North

MANNING

BDD in Action

by John Ferguson Smart

Chapter 1

brief contents

1 Building software that makes a difference

This chapter covers

- The problems that Behavior-Driven Development addresses
- General principles and origins of Behavior-Driven Development
- Activities and outcomes seen in a Behavior-Driven Development project
- The pros and cons of Behavior-Driven Development

This book is about building and delivering better software. It's about building software that works well and is easy to change and maintain, but more importantly, it's about building software that provides real value to its users. We want to build software well, but we also need to build software that's worth building.

In 2012, the U.S. Air Force decided to ditch a major software project that had already cost over $1 billion USD. The Expeditionary Combat Support System was designed to modernize and streamline supply chain management in order to save billions of dollars and meet new legislative requirements. But after seven years of development, the system had still "not yielded any significant military capability."[1]

[1] Chris Kanaracus, "Air Force scraps massive ERP project after racking up $1 billion in costs," *CIO*, November 14, 2012, http://www.cio.com/article/2390341.

The Air Force estimated that an additional $1.1 billion USD would be required to deliver just a quarter of the original scope, and that the solution could not be rolled out until 2020, three years after the legislative deadline of 2017.

This happens a lot in the software industry. According to a number of studies, around half of all software projects fail to deliver in some significant way. The 2011 edition of the Standish Group's annual *CHAOS Report* found that 42% of projects were delivered late, ran over budget, or failed to deliver all of the requested features,[2] and 21% of projects were cancelled entirely. Scott Ambler's annual survey on IT project success rates uses a more flexible definition of success, but still found a 30–50% failure rate, depending on the methodologies used.[3] This corresponds to billions of dollars in wasted effort, writing software that ultimately won't be used or that doesn't solve the business problem it was intended to solve.

What if it didn't have to be this way? What if we could write software in a way that would let us discover and focus our efforts on what really matters? What if we could objectively learn what features will really benefit the organization and the most cost-effective way to implement them? What if we could see beyond what the user asks for and build what the user actually needs?

There are organizations discovering how to do just that. Many teams are successfully collaborating to build and deliver more valuable, more effective, and more reliable software. And they're learning to do this faster and more efficiently. In this book, you'll see how—we'll explore a number of methods and techniques, grouped under the general heading of *Behavior-Driven Development* (BDD).

BDD helps teams focus their efforts on identifying, understanding, and building valuable features that matter to businesses, and it makes sure that these features are well designed and well implemented.

BDD practitioners use conversations around concrete examples of system behavior to help understand how features will provide value to the business. BDD encourages business analysts, software developers, and testers to collaborate more closely by enabling them to express requirements in a more testable way, in a form that both the development team and business stakeholders can easily understand. BDD tools can help turn these requirements into automated tests that help guide the developer, verify the feature, and document what the application does.

BDD isn't a software development methodology in its own right. It's not a replacement for Scrum, XP, Kanban, RUP, or whatever methodology you're currently using. As you'll see, BDD incorporates, builds on, and enhances ideas from many of these methodologies. And no matter what methodology you're using, there are ways that BDD can help make your life easier.

---

[2] Whether these figures reflect more on our ability to build and deliver software or on our ability to plan and estimate is a subject of some debate in the Agile development community—see Jim Highsmith's book *Agile Project Management: Creating Innovative Products*, second edition (Addison-Wesley Professional, 2009).

[3] Scott Ambler, Surveys Exploring the Current State of Information Technology Practices, http://www.ambysoft.com/surveys/.

1.1 BDD from 50,000 feet

So what does BDD bring to the table? Here's a (slightly oversimplified) perspective. Let's say Chris's company needs a new module for its accounting software. When Chris wants to add a new feature, the process goes something like this (see figure 1.1):

1. Chris tells a business analyst how he would like the feature to work.
2. The business analyst translates Chris's requests into a set of requirements for the developers, describing what the software should do. These requirements are written in English and stored in a Microsoft Word document.
3. The developer translates the requirements into code and unit tests—written in Java, C#, or some other programming language—in order to implement the new feature.
4. The tester translates the requirements in the Word document into test cases, and uses them to verify that the new feature meets the requirements.
5. Documentation engineers then translate the working software and code back into plain English technical and functional documentation.

There are many opportunities for information to get lost in translation, be misunderstood, or just be ignored. Chances are that the new module itself may not do exactly what was required and that the documentation won't reflect the initial requirements that Chris gave the analyst.

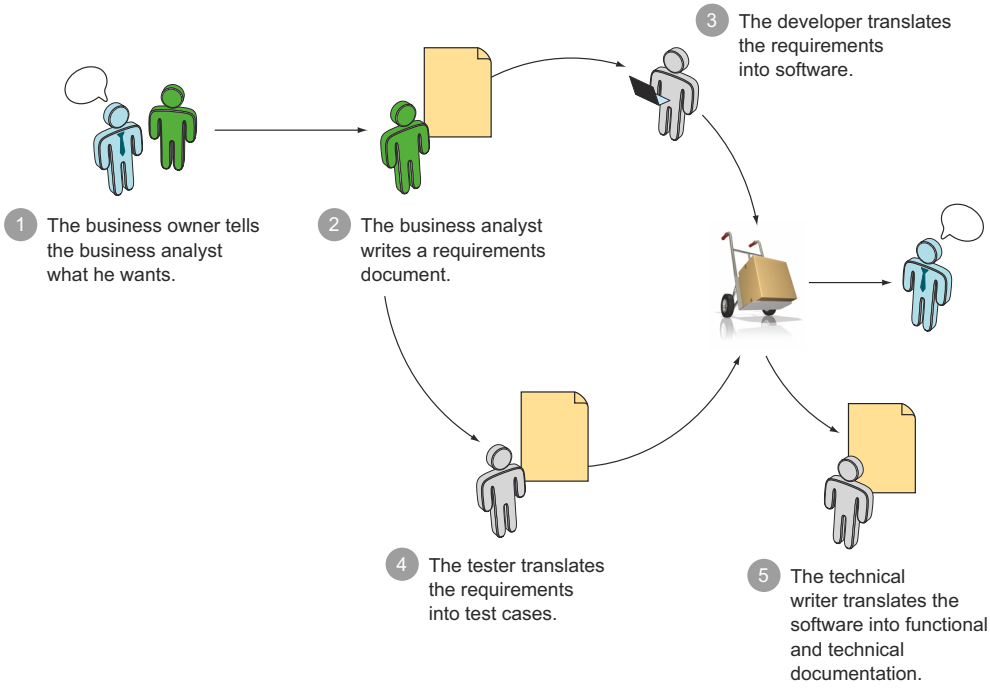

Figure 1.1 The traditional development process provides many opportunities for misunderstandings and miscommunication.

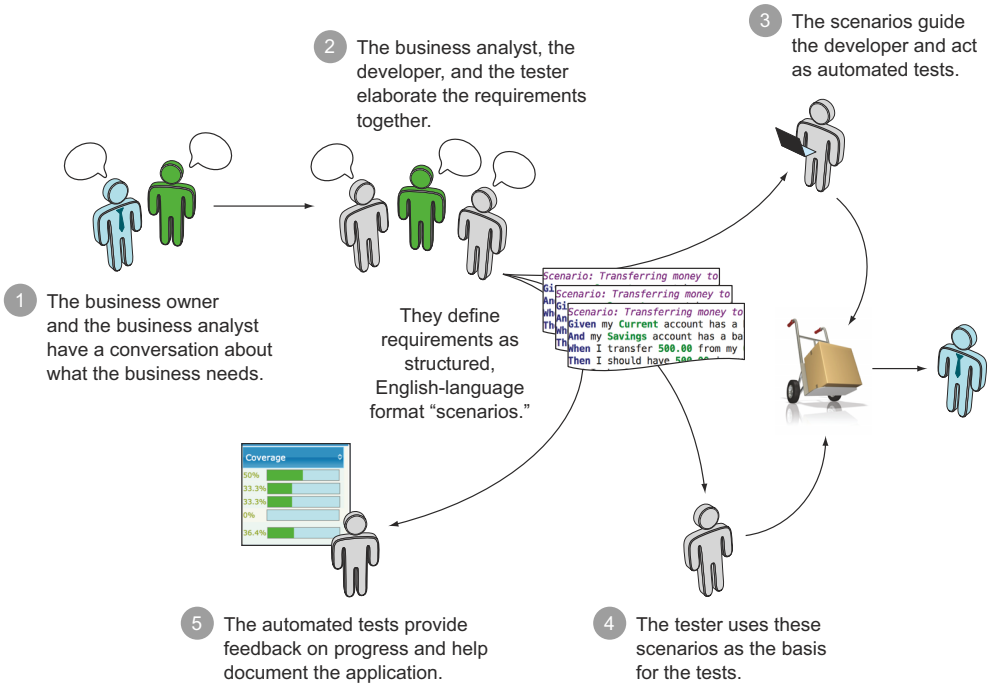

Figure 1.2 BDD uses conversations around examples, expressed In a form that can be easily automated, to reduce lost information and misunderstandings.

Chris's friend Sarah runs another company that just introduced BDD. In a team practicing BDD, the business analysts, developers, and testers collaborate to understand and define the requirements together (see figure 1.2). They use a common language that allows for an easy, less ambiguous path from end-user requirements to usable, automatable tests. These tests specify how the software should behave, and they guide the developers in building working software that focuses on features that really matter to the business.

1. Like Chris, Sarah talks to a business analyst about what she wants. To reduce the risk of misunderstandings and hidden assumptions, they talk through concrete examples of what the feature should do.
2. Before work starts on the feature, the business analyst gets together with the developer and tester who will be working on it, and they have a conversation about the feature. In this conversation, they discuss and translate key examples of how the feature should work into a set of requirements written in a structured, English-language format often referred to as Gherkin.
3. The developer uses a BDD tool to turn these requirements into a set of automated tests that run against the application code and help objectively determine when a feature is finished.

4. The tester uses the results of these tests as the starting point for manual and exploratory tests.
5. The automated tests act as low-level technical documentation, and provide up-to-date examples of how the system works. Sarah can review the test reports to see what features have been delivered, and whether they work the way she expected.

Compared to Chris's scenario, Sarah's team makes heavy use of conversations and examples to reduce the amount of information lost in translation. Every stage beyond step 2 starts with the specifications written in Gherkin, which are based on concrete examples provided by Sarah. In this way, a great deal of the ambiguity in translating the client's initial requirements into code, reports, and documentation is removed.

We'll discuss all of these points in detail throughout the rest of the book. You'll learn ways to help ensure that your code is of high quality, solid, well tested, and well documented. You'll learn how to write more effective unit tests and more meaningful automated acceptance criteria. You'll also learn how to ensure that the features you deliver solve the right problems and provide real benefit to the users and the business.

1.2 What problems are you trying to solve?

Software projects fail for many reasons, but the most significant causes fall into two broad categories:

- Not building the software right
- Not building the right software

Figure 1.3 illustrates this in the form of a graph. The vertical axis represents *what* you're building, and the horizontal axis represents *how* you build it. If you perform poorly on the *how* axis, not writing well-crafted and well-designed software, you'll end up with a buggy, unreliable product that's hard to change and maintain. If you don't do well on the *what* axis, failing to understand what features the business really needs, you'll end up with a product that nobody needs.

1.2.1 Building the software right

Many projects suffer or fail because of software quality issues. Although internal software quality is mostly invisible to nontechnical stakeholders, the consequences of poor-quality software can be painfully visible. In my experience, applications that are poorly designed, badly written, or lack well-written, automated tests tend to be buggy, hard to maintain, hard to change, and hard to scale.

I've seen too many applications where simple change requests and new features take too long to deliver. Developers spend more and more time fixing bugs rather than working on new features, which makes it harder to deliver new features quickly. It takes longer for new developers to get up to speed and become productive, simply because the code is hard to understand. It also becomes harder and harder to add new features without breaking existing code. The existing technical documentation (if there is any) is inevitably out of date, and teams find themselves incapable of

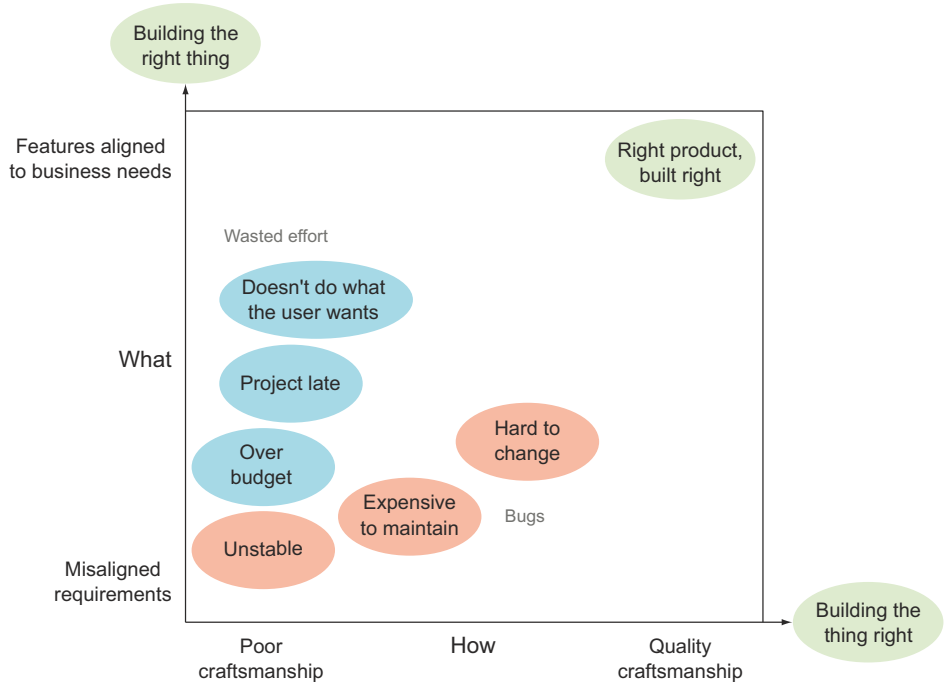

Figure 1.3 Successful projects must both build features well and build the right features.

delivering new features quickly because each release requires a lengthy period of manual testing and bug fixes.

Organizations that embrace high-quality technical practices have a different story to tell. I've seen many teams that adopt practices such as Test-Driven Development, Clean Coding, Living Documentation, and Continuous Integration regularly reporting low to near-zero defect rates, as well as code that's much easier to adapt and extend as new requirements emerge and new features are requested. These teams can also add features at a more consistent pace, because the automated tests ensure that existing features won't be broken unknowingly. They implement the features faster and more precisely than other teams because they don't have to struggle with long bug-fixing sessions and unpredictable side effects when they make changes. And the resulting application is easier and cheaper to maintain.

Note that there is no magic formula for building high-quality, easily maintainable software. Software development is a complex field, human factors abound, and techniques such as Test-Driven Development, Clean Coding, and Automated Testing don't automatically guarantee good results. But studies do suggest a strong correlation between lean and Agile practices and project success rates[4] when compared to more

[4] See, for example, Ambysoft, "2013 IT Project Success Rates Survey Results," http://www.ambysoft.com/surveys/success2013.html.

traditional approaches. Other studies have found a correlation between Test-Driven Development practices, reduced bug counts,[5] and improved code quality.[6] Although it's certainly possible to write high-quality code without practicing techniques such as Test-Driven Development and Clean Coding, teams that value good development practices do seem to succeed in delivering high-quality code more often.

But building high-quality software isn't in itself enough to guarantee a successful project. The software must also benefit its users and business stakeholders.

1.2.2 Building the right software

Software is never developed in a vacuum. Software projects are part of a broader business strategy, and they need to be aligned with business goals if they're to be beneficial to the organization. At the end of the day, the software solution you deliver needs to help users achieve their goals more effectively. Any effort that doesn't contribute to this end is waste.

In practice, there's often a lot of waste. In many projects, time and money are spent building features that are never used or that provide only marginal value to the business. According to the Standish Group's CHAOS studies,[7] on average some 45% of the features delivered into production are never used. Even apparently predictable projects, such as migrating software from a mainframe system onto a more modern platform, have their share of features that need updating or that are no longer necessary. When you don't fully understand the goals that your client is trying to achieve, it's very easy to deliver perfectly functional, well-written features that are of little use to the end user.

On the other hand, many software projects end up delivering little or no real business value. Not only do they deliver features that are of little use to the business, but they fail to even deliver the minimum capabilities that would make the projects viable.

The consequences of not building it right, and not building the right thing

The impact of poorly understood requirements and poor code realization isn't just a theoretical concept or a "nice to have;" on the contrary, it's often painfully concrete. In December 2007, the Queensland Health Department kicked off work on a new payroll system for its 85,000 employees. The initial budget for the project was around $6 million, with a delivery date of August 2008.

---

[5] See, for example, Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf.

[6] Rod Hilton, "Quantitatively Evaluating Test-Driven Development by Applying Object-Oriented Quality Metrics to Open Source Projects" (PhD thesis, Regis University, 2009), http://www.rodhilton.com/files/tdd_thesis.pdf.

[7] The Standish Group's *CHAOS Report 2002* reported a value of 45% and I've seen more recent internal studies where the figure is around 50%.

(continued)

When the solution was rolled out in 2010, some 18 months late, it was a disaster.[8] Tens of thousands of public servants were underpaid, overpaid, or not paid at all. Since the go-live date, over 1,000 payroll staff have been required to carry out some 200,000 manual processes each fortnight to ensure that staff salaries are paid.

In 2012, an independent review found that the project had cost the state over $416 million since going into production and would cost an additional $837 million to fix. This colossal sum included $220 million just to fix the immediate software issues that were preventing the system from delivering its core capability of paying Queensland Health staff what they were owed each month.

Building the right software is made even trickier by one commonly overlooked fact: early on in a project, you usually don't know what the right features are.

1.2.3 The knowledge constraint—dealing with uncertainty

One fact of life in software development is that there will be things you don't know. Changing requirements are a normal part of every software project. Knowledge and understanding about the problem at hand and about how best to solve it increases progressively throughout the project.

In software development, each project is different. There are always new business requirements to cater to, new technological problems to solve, and new opportunities to seize. As a project progresses, market conditions, business strategies, technological constraints, or simply your understanding of the requirements will evolve, and you'll need to change your tack and adjust your course. Each project is a journey of discovery where the real constraint isn't time, the budget, or even programmer hours, but your lack of knowledge about what you need to build and how you should build it. When reality doesn't go according to plan, you need to adapt to reality, rather than trying to force reality to fit into your plan. "When the terrain disagrees with the map, trust the terrain" (Swiss Army proverb).

Users and stakeholders will usually know what high-level goals they want to achieve and can be coaxed into revealing these goals if you take the time to ask. They'll be able to tell you that they need an online ticketing system or a payroll solution that caters to 85,000 different employees. And you can get a feel for the scope of the application you might need to build early on in the project.

But the details are another matter entirely. Although users are quick to ask for specific technical solutions to their problems, they're not usually the best-placed to know what solution would serve them best, or even, for that matter, what solutions exist. Your team's collective understanding of the best way to deliver these capabilities, as

[8] See KPMG, "Review of the Queensland Health Payroll System" (2012), http://delimiter.com.au/wp-content/uploads/2012/06/KPMG_audit.pdf.

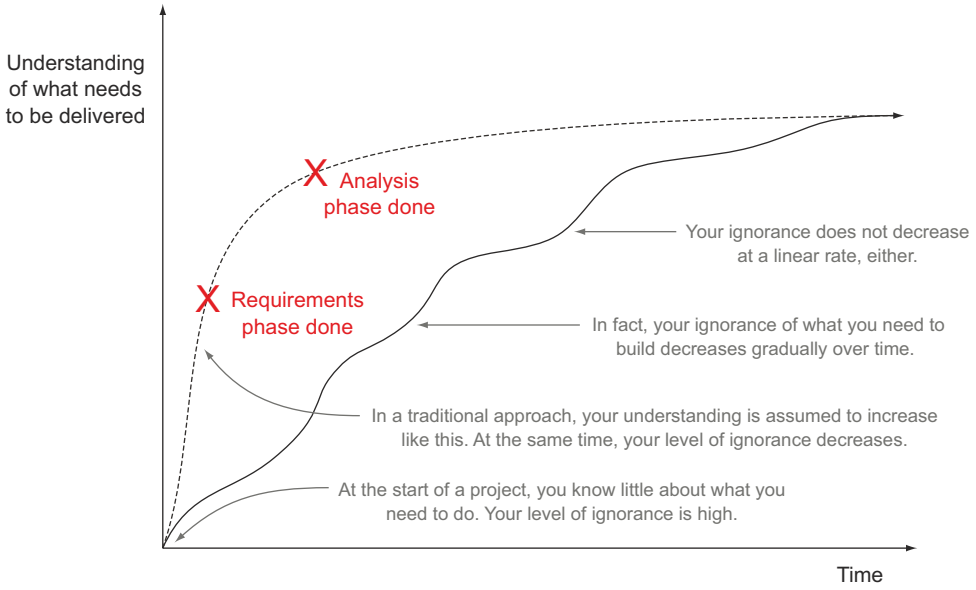

Figure 1.4 At the start of a project, there are many unknowns. You reduce these unknowns as the project progresses, but not in a linear or very predictable way.

well as the optimal feature set for achieving the underlying business goals, will grow as the project progresses.

As illustrated in figure 1.4, the more prescriptive, plan-based requirements-analysis techniques suppose that you can learn almost all there is to know about a project's requirements, as well as the optimal solution design, very quickly in the early phases of the project. By the end of the analysis phase, the specifications are signed-off on and locked down, and all that remains to do is code.

Of course, reality doesn't always work this way. At the start of the project, a development team will often have only a superficial understanding of the business domain and the goals the users need to achieve. In fact, the job of a software engineering team isn't to know how to build a solution; it's to know how to discover the best way to build the solution.

The team's collective understanding will naturally increase over the duration of the project. You become less ignorant over time. Toward the end of the project, a good team will have built up a deep, intimate knowledge of the user's needs and will be able to proactively propose features and implementations that will be better suited to the particular user base. But this learning path is neither linear nor predictable. It's hard to know what you don't know, so it's hard to predict what you'll learn as the project progresses.

For the majority of modern software development projects, the main challenge in managing scope isn't to eliminate uncertainty by defining and locking down requirements as early as possible. The main challenge is to manage this uncertainty in a way

that will help you progressively discover and deliver an effective solution that matches up with the underlying business goals behind a project. As you'll see, one important benefit of BDD is that it provides techniques that can help you manage this uncertainty and reduce the risk that comes with it.

1.3 Introducing Behavior-Driven Development

Behavior-Driven Development (BDD) is a set of software engineering practices designed to help teams build and deliver more valuable, higher quality software faster. It draws on Agile and lean practices including, in particular, Test-Driven Development (TDD) and Domain-Driven Design (DDD). But most importantly, BDD provides a common language based on simple, structured sentences expressed in English (or in the native language of the stakeholders) that facilitate communication between project team members and business stakeholders.

To better understand the motivations and philosophy that drive BDD practices, it's useful to understand where BDD comes from.

1.3.1 BDD was originally designed as an improved version of TDD

BDD was originally invented by Dan North[9] in the early to mid-2000s as an easier way to teach and practice Test-Driven Development (TDD). TDD, invented by Kent Beck in the early days of Agile,[10] is a remarkably effective technique that uses unit tests to specify, design, and verify application code.

When TDD practitioners need to implement a feature, they first write a failing test that describes, or specifies, that feature. Next, they write just enough code to make the test pass. Finally, they refactor the code to help ensure that it will be easy to maintain (see figure 1.5). This simple but powerful technique encourages developers to write cleaner, better-designed, easier-to-maintain code[11] and results in substantially lower defect counts.[12]

Despite its advantages, many teams still have difficulty adopting and using TDD effectively. Developers often have trouble knowing where to start or what tests they should write next. Sometimes TDD can lead developers to become too detail-focused, losing the broader picture of the business goals they're supposed to implement. Some teams also find that the large numbers of unit tests can become hard to maintain as the project grows in size.

In fact, many traditional unit tests, written with or without TDD, are tightly coupled to a particular implementation of the code. They focus on the method or function they're testing, rather than on what the code should do in business terms.

---

[9] Dan North, "Introducing BDD," http://dannorth.net/introducing-bdd/.

[10] Kent Beck, *Test-Driven Development: By Example* (Addison-Wesley Professional, 2002).

[11] Rod Hilton, "Quantitatively Evaluating Test-Driven Development by Applying Object-Oriented Quality Metrics to Open Source Projects" (PhD thesis, Regis University, 2009), http://www.rodhilton.com/files/tdd_thesis.pdf.

[12] Nachiappan Nagappan et al., "Realizing Quality Improvement through Test Driven Development" (2008), http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf.

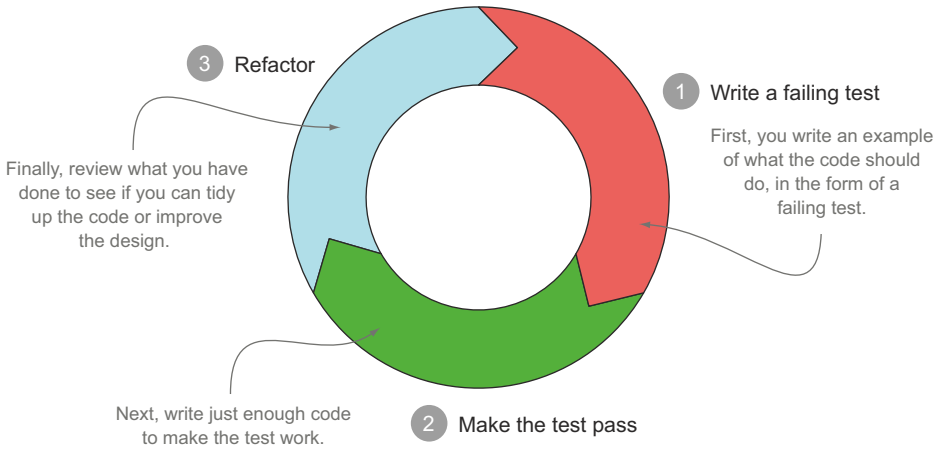

Figure 1.5 Test-Driven Development relies on a simple, three-phase cycle.

For example, suppose Paul is a Java developer working on a new financial trading application in a large bank. He has been asked to implement a new feature to transfer money from one account to another. He creates an `Account` class with a `transfer()` method, a `deposit()` method, and so on. The corresponding unit tests are focused on testing these methods:

```
public class BankAccountTest {
    @Test
    public void testTransfer() {...}

    @Test
    public void testDeposit() {...}
}
```

Tests like this are better than nothing, but they can limit your options. For example, they don't describe what you expect the `transfer()` and `deposit()` functions to do, which makes them harder to understand and to fix if they break. They're tightly coupled to the method they test, which means that if you refactor the implementation, you need to rename your test as well. And because they don't say much about what they're actually testing, it's hard to know what other tests (if any) you need to write before you're done.

North observed that a few simple practices, such as naming unit tests as full sentences and using the word "should," can help developers write more meaningful tests, which in turn helps them write higher quality code more efficiently. When you think in terms of what the class *should* do, instead of what method or function is being tested, it's easier to keep your efforts focused on the underlying business requirements.

For example, Paul could write more descriptive tests along the following lines:

```
public class WhenTransferringInternationalFunds {
    @Test
    public void should_transfer_funds_to_a_local_account() {...}
```

```
    @Test
    public void should_transfer_funds_to_a_different_bank() {...}
    ...

    @Test
    public void should_deduct_fees_as_a_separate_transaction() {...}
    ...
}
```

Tests that are written this way read more like specifications than unit tests. They focus on the behavior of the application, using tests simply as a means to express and verify that behavior. North also noted that tests written this way are much easier to maintain because their intent is so clear. The impact of this approach was so significant that he started referring to what he was doing no longer as Test-Driven Development, but as *Behavior*-Driven Development.

1.3.2 BDD also works well for requirements analysis

But describing a system's behavior turns out to be what business analysts do every day. Working with business analyst colleague Chris Matts, North set out to apply what he had learned to the requirements-analysis space. Around this time, Eric Evans introduced the idea of Domain-Driven Design,[13] which promotes the use of a ubiquitous language that business people can understand to describe and model a system. North and Matts's vision was to create a ubiquitous language that business analysts could use to define requirements unambiguously, and that could also be easily transformed into automated acceptance tests. To implement this vision, they started expressing the acceptance criteria for user stories in the form of loosely structured examples, known as "scenarios," like this one:

```
Given a customer has a current account
When the customer transfers funds from this account to an overseas account
Then the funds should be deposited in the overseas account
And the transaction fee should be deducted from the current account
```

A business owner can easily understand a scenario written like this. It gives clear and objective goals for each story in terms of what needs to be developed and of what needs to be tested.

This notation eventually evolved into a commonly used form often referred to as Gherkin. With appropriate tools, scenarios written in this form can be turned into automated acceptance criteria that can be executed automatically whenever required. Dan North wrote the first dedicated BDD test automation library, JBehave, in the mid-2000s, and since then many others have emerged for different languages, both at the unit-testing and acceptance-testing levels.

[13] Eric Evans, *Domain Driven Design* (Addison-Wesley Professional, 2003).

BDD by any other name

Many of the ideas around BDD are not new and have been practiced for many years under a number of different names. Some of the more common terms used for these practices include Acceptance-Test-Driven Development, Acceptance Test-Driven Planning, and Specification by Example. To avoid confusion, let's clarify a few of these terms in relation to BDD.

Specification by Example describes the set of practices that have emerged around using examples and conversation to discover and describe requirements. In his seminal book of the same name,[14] Gojko Adzic chose this term as the most representative name to refer to these practices. Using conversation and examples to specify how you expect a system to behave is a core part of BDD, and we'll discuss it at length in the first half of this book.

Acceptance-Test-Driven Development (ATDD) is now a widely used synonym for Specification by Example, but the practice has existed in various forms since at least the late 1990s. Kent Beck and Martin Fowler mentioned the concept in 2000,[15] though they observed that it was difficult to implement acceptance criteria in the form of conventional unit tests at the start of a project. But unit tests aren't the only way to write automated acceptance tests, and since at least the early 2000s, innovative teams have asked users to contribute to executable acceptance tests and have reaped the benefits.[16]

Acceptance-Test-Driven Planning is the idea that defining acceptance criteria for a feature leads to better estimates than doing a task breakdown.

1.3.3 BDD principles and practices

Today BDD is successfully practiced in a large number of organizations of all sizes around the world, in a variety of different ways. In *Specification by Example*, Gojko Adzic provides case studies for over 50 such organizations. In this section, we'll look at a number of general principles or guidelines that BDD practitioners have found useful over the years.

Figure 1.6 gives a high-level overview of the way BDD sees the world. BDD practitioners like to start by identifying business goals and looking for features that will help deliver these goals. Collaborating with the user, they use concrete examples to illustrate these features. Wherever possible, these examples are automated in the form of executable specifications, which both validate the software and provide automatically updated technical and functional documentation. BDD principles are also used at the coding level, where they help developers write code that's of higher quality, better tested, better documented, and easier to use and maintain.

---

[14] Gojko Adzik, *Specification by Example* (Manning, 2011).

[15] Kent Beck and Martin Fowler, *Planning Extreme Programming* (Addison-Wesley Professional, 2000).

[16] Johan Andersson et al., "XP with Acceptance-Test Driven Development: A Rewrite Project for a Resource Optimization System," *Lecture Notes in Computer Science Volume* 2675 (2003). Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.6097&rep=rep1&type=pdf.

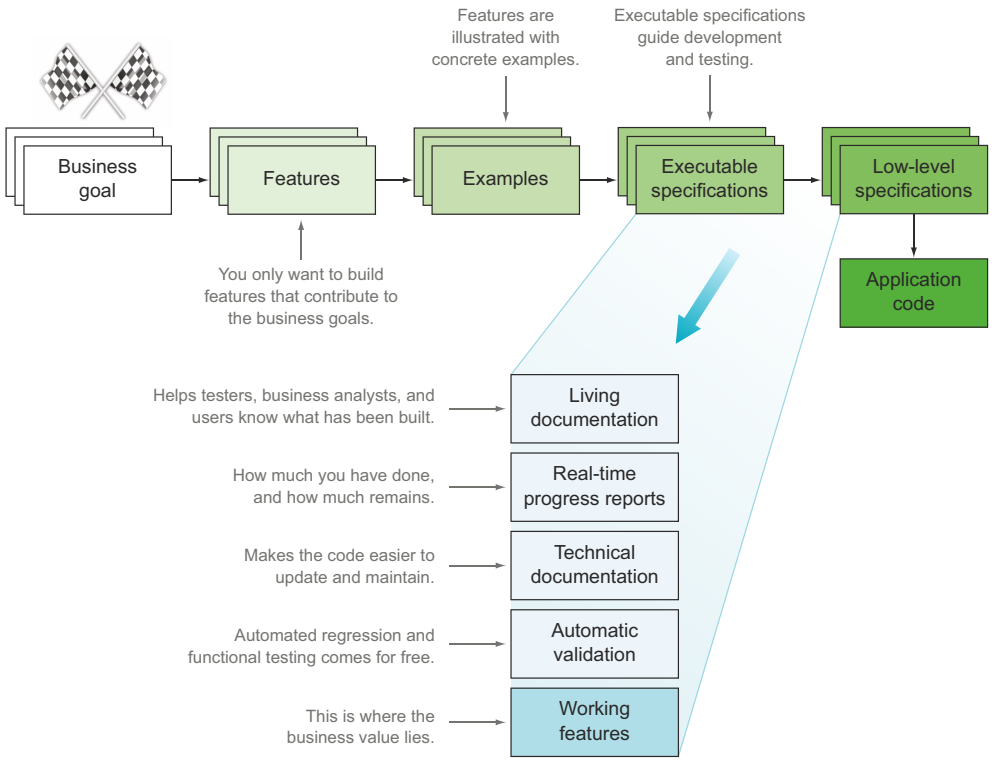

Figure 1.6 The principal activities and outcomes of BDD. Note that these activities occur repeatedly and continuously throughout the process; this isn't a single linear Waterfall-style process, but a sequence of activities that you practice for each feature you implement.

In the following sections, we'll look at how these principles work in more detail.

FOCUS ON FEATURES THAT DELIVER BUSINESS VALUE

As you've seen, uncertainty about requirements is a major challenge in many software projects, and heavy upfront specifications don't work particularly well when confronted with a shifting understanding of what features need to be delivered.

A *feature* is a tangible, deliverable piece of functionality that helps the business to achieve its business goals. For example, suppose you work in a bank that's implementing an online banking solution. One of the business goals for this project might be "to attract more clients by providing a simple and convenient way for clients to manage their accounts." Some features that might help achieve this goal could be "Transfer funds between a client's accounts," "Transfer funds to another national account," or "Transfer funds to an overseas account."

Rather than attempting to nail down all of the requirements once and for all, teams practicing BDD engage in ongoing conversations with the end users and other stakeholders to progressively build a common understanding of what features they should create. Rather than working upfront to design a complete solution for

the developers to implement, users explain what they need to get out of the system and how it might help them achieve their objectives. And rather than accepting a list of feature requests from the users with no questions asked, teams try to understand the core business goals underlying the project, proposing only features that can be demonstrated to support these business goals. This constant focus on delivering business value means that teams can deliver more useful features earlier and with less wasted effort.

WORK TOGETHER TO SPECIFY FEATURES

> A complex problem, like discovering ways to delight clients, is best solved by a cognitively diverse group of people that is given responsibility for solving the problem, self-organizes, and works together to solve it.
>
> Stephen Denning, *The Leader's Guide to Radical Management* (Jossey-Bass, 2010)

BDD is a highly collaborative practice, both between users and the development team, and within the team itself. Business analysts, developers, and testers work together with the end users to define and specify features, and team members draw ideas from their individual experience and know-how. This approach is highly efficient.

In a more traditional approach, when business analysts simply relay their understanding of the users' requirements to the rest of the team, there is a high risk of misinterpretation and lost information.

If you ask users to write up what they want, they'll typically give you a set of detailed requirements that matches how they envisage the solution. In other words, users will not tell you *what they need*; rather, *they'll design a solution for you*. I've seen many business analysts fall into the same trap, simply because they've been trained to write specifications that way. The problem with this approach is twofold: not only will they fail to benefit from the development team's expertise in software design, but they're effectively binding the development team to a particular solution, which may not be the optimal one in business or technical terms. In addition, developers can't use their technical know-how to help deliver a technically superior design, and testers don't get the opportunity to comment on the testability of the specifications until the end of the project.

For example, the "Transfer funds to an overseas account" feature involves many user-experience and technical considerations. How can you display the constantly changing exchange rates to the client? When and how are the fees calculated and shown to the client? For how long can you guarantee a proposed exchange rate? How can you verify that the right exchange rate is being used? All of these considerations will have an impact on the design, implementation, and cost of the feature and can change the way the business analysts and business stakeholders originally imagined the solution.

When teams practice BDD, on the other hand, team members build up a shared appreciation of the users' needs, as well as a sense of common ownership and engagement in the solution.

EMBRACE UNCERTAINTY

A BDD team knows that they won't know everything upfront, no matter how long they spend writing specifications. As we discussed earlier, the biggest thing slowing developers down in a software project is understanding what they need to build.

Rather than attempting to lock down the specifications at the start of the project, BDD practitioners assume that the requirements, or more precisely, their *understanding of* the requirements, will evolve and change throughout the life of a project. They try to get early feedback from the users and stakeholders to ensure that they're on track, and change tack accordingly, instead of waiting until the end of the project to see if their assumptions about the business requirements were correct.

Very often, the most effective way to see if users like a feature is to build it and show it to them as early as possible. With this in mind, experienced BDD teams prioritize the features that will deliver value, will improve their understanding of what features the users really need, and will help them understand how best to build and deliver these features.

ILLUSTRATE FEATURES WITH CONCRETE EXAMPLES

When a team practicing BDD decides to implement a feature, they work together with users and other stakeholders to define stories and scenarios of what users expect this feature to deliver. In particular, the users help define a set of concrete examples that illustrate key outcomes of the feature (see figure 1.7).

These examples use a common vocabulary and can be readily understood by both end users and members of the development team. They're usually expressed using the *Given ... When ... Then* notation you saw in section 1.3.2. For instance, a simple example that illustrates the "Transfer funds between a client's accounts" feature might look like this:

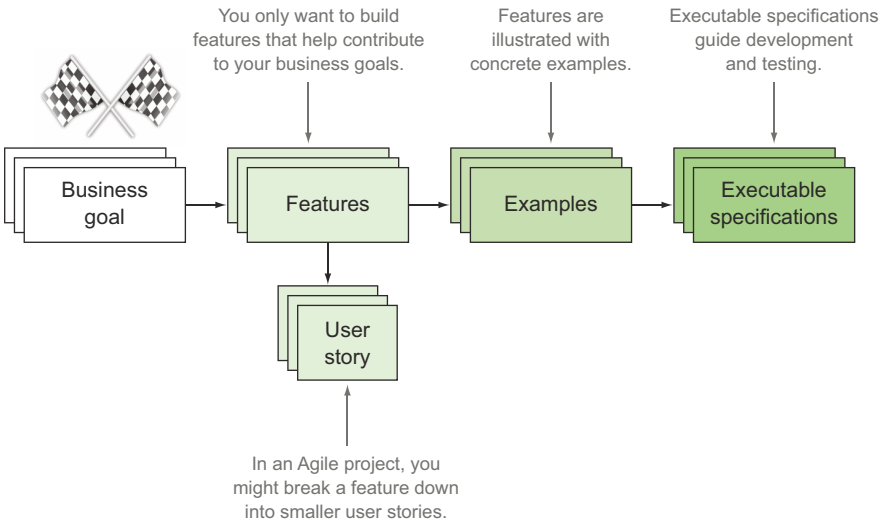

Figure 1.7 Examples play a primary role in BDD, helping everyone understand the requirements more clearly.

```
Scenario: Transferring money to a savings account
    Given I have a current account with 1000.00
    And I have a savings account with 2000.00
    When I transfer 500.00 from my current account to my savings account
    Then I should have 500.00 in my current account
    And I should have 2500.00 in my savings account
```

Examples play a primary role in BDD, simply because they're an extremely effective way of communicating clear, precise, and unambiguous requirements. Specifications written in natural language are, as it turns out, a terribly poor way of communicating requirements, because there's so much space for ambiguity, assumptions, and misunderstandings. Examples are a great way to overcome these limitations and clarify the requirements.

Examples are also a great way to explore and expand your knowledge. When a user proposes an example of how a feature should behave, project team members often ask for extra examples to illustrate corner cases, explore edge cases, or clarify assumptions. Testers are particularly good at this, which is why it's so valuable for them to be involved at this stage of the project.

A Gherkin primer

Most BDD tools that we'll look at in this book use a format generally known as Gherkin, or a very close variation on this format used by JBehave.[17] This format is designed to be both easily understandable for business stakeholders and easy to automate using dedicated BDD tools such as Cucumber and JBehave. This way, it both documents your requirements and runs your automated tests.

In Gherkin, the requirements related to a particular feature are grouped into a single text file called a *feature file*. A feature file contains a short description of the feature, followed by a number of scenarios, or formalized examples of how a feature works.

```
Feature: Transferring money between accounts
  In order to manage my money more efficiently
  As a bank client
  I want to transfer funds between my accounts whenever I need to

  Scenario: Transferring money to a savings account
    Given my Current account has a balance of 1000.00
    And my Savings account has a balance of 2000.00
    When I transfer 500.00 from my Current account to my Savings account
    Then I should have 500.00 in my Current account
    And I should have 2500.00 in my Savings account
  Scenario: Transferring with insufficient funds
    Given my Current account has a balance of 1000.00
    And my Savings account has a balance of 2000.00
    When I transfer 1500.00 from my Current account to my Savings account
    Then I should receive an 'insufficient funds' error
    Then I should have 1000.00 in my Current account
    And I should have 2000.00 in my Savings account
```

[17] Strictly speaking, Gherkin refers to the format recognized by the Cucumber family of BDD automation tools (see http://cukes.info). For simplicity, we'll use the term *Gherkin* to refer to both variations, and I'll indicate any differences as we come across them.

(continued)

As can be seen here, Gherkin requirements are expressed in plain English, but with a specific structure. Each *scenario* is made up of a number of *steps*, where each step starts with one of a small number of keywords (*Given*, *When*, *Then*, *And*, and *But*).

The natural order of a scenario is *Given ... When ... Then*:

- *Given* describes the preconditions for the scenario and prepares the test environment.
- *When* describes the action under test.
- *Then* describes the expected outcomes.

The *And* and *But* keywords can be used to join several *Given*, *When*, or *Then* steps together in a more readable way:

```
Given I have a current account with $1000
And I have a savings account with $2000
```

Several related scenarios can often be grouped into a single scenario using a table of examples. For example, the following scenario illustrates how interest is calculated on different types of accounts:

```
Scenario Outline: Earning interest
  Given I have an account of type <account-type> with a balance of
     <initial-balance>
  When the monthly interest is calculated
  Then I should have earned at an annual interest rate of <interest-rate>
  And I should have a new balance of <new-balance>
  Examples:
  | initial-balance | account-type | interest-rate | new-balance
  | 10000           | current      | 1             | 10008.33
  | 10000           | savings      | 3             | 10025
  | 10000           | supersaver   | 5             | 10041.67
```

This scenario would be run three times in all, once for each row in the Examples table. The values in each row are inserted into the placeholder variables, which are indicated by the `<...>` notation (`<account-type>`, `<initial-balance>`, and so forth). This not only saves typing, but also makes it easier to understand the whole requirement at a glance.

You can also use the following tabular notation within the steps themselves in order to display test data more concisely. For example, the previous money-transfer scenario could have been written like this:

```
Scenario: Transferring money between accounts within the bank
  Given I have the following accounts:
    | account | balance |
    | current | 1000    |
    | savings | 2000    |
  When I transfer 500.00 from current to savings
  Then my accounts should look like this:
    | account | balance |
    | current | 500     |
    | savings | 2500    |
```

We'll look at this notation in much more detail in chapter 5.

DON'T WRITE AUTOMATED TESTS, WRITE EXECUTABLE SPECIFICATIONS

These stories and examples form the basis of the specifications that developers use to build the system. They act as both acceptance criteria, determining when a feature is done, and as guidelines for developers, giving them a clear picture of what needs to be built.

Acceptance criteria give the team a way to objectively judge whether a feature has been implemented correctly. But checking this manually for each code change would be time-consuming and inefficient. It would also slow down feedback, which would in turn slow down the development process. Wherever feasible, teams turn these acceptance criteria into automated acceptance tests or, more precisely, into *executable specifications*.

An executable specification is an automated test that illustrates and verifies how the application delivers a specific business requirement. These automated tests run as part of the build process and run whenever a change is made to the application. In this way, they serve both as acceptance tests, determining which new features are complete, and as regression tests, ensuring that new changes haven't broken any existing features (see figure 1.8).

You can automate an executable specification by writing test code corresponding to each step. BDD tools like Cucumber and JBehave will match the text in each step of your scenario to the appropriate test code.

For example, this is the first step of the scenario in figure 1.8:

```
Given my Current account has a balance of 1000.00
```

You might automate this step in Java using JBehave with code like this:

```
@Given("my $accountType account has a balance of $amount")
public void setupInitialAccount(AccountType type, double amount) {
    Account account = Account.ofType(type).withInitialBalance(amount);
    accountService.create(account);
}
```

❶ **The step that this code implements**

Call the application code that corresponds to this step ❷

When JBehave runs the scenario, it'll execute each step of the scenario, using basic pattern matching to find the method associated with this step ❶. Once it knows what method to call, it'll extract variables like `type` and `amount` and execute the corresponding application code ❷.

Unlike conventional unit or integration tests, or the automated functional tests many QA teams are used to, executable specifications are expressed in something close to natural language. They use precisely the examples that the users and development team members proposed and refined earlier on, using exactly the same terms and vocabulary. Executable specifications are about communication as much as they are about validation, and the test reports they generate are easily understandable by everyone involved with the project.

These executable specifications also become a single source of truth, providing reference documentation for how features should be implemented. This makes

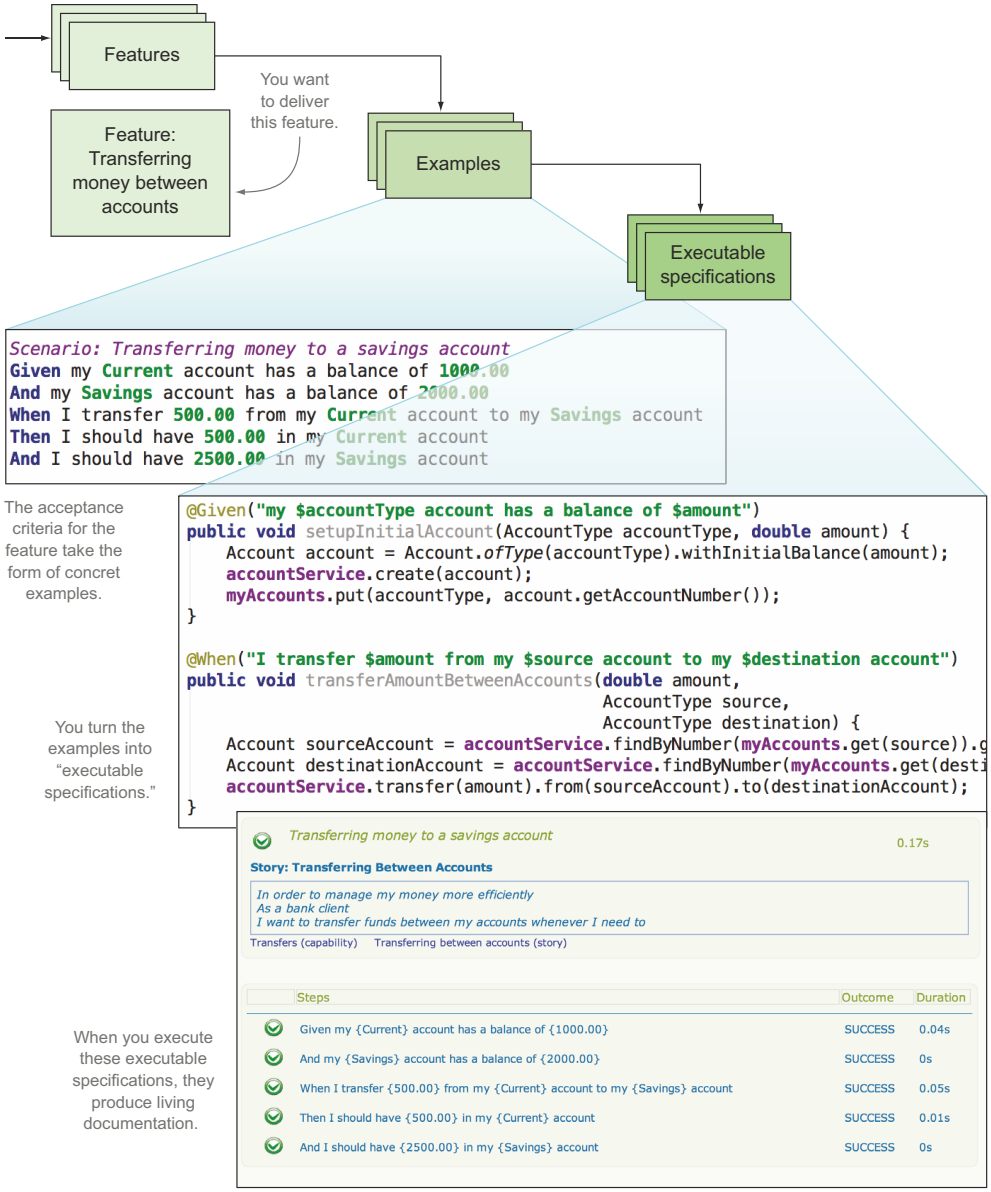

Figure 1.8 Executable specifications are expressed using a common business vocabulary that the whole team can understand. They guide development and testing activities and produce readable reports available to all.

maintaining the requirements much easier. If specifications are stored in the form of a Word document or on a Wiki page, as is done for many traditional projects, any changes to the requirements need to be reflected both in the requirements document and in the acceptance tests and test scripts, which introduces a high risk of

inconsistency. For teams practicing BDD, the requirements and executable specifications are the same thing; when the requirements change, the executable specifications are updated directly in a single place. We'll look at this in detail in chapter 9.

DON'T WRITE UNIT TESTS, WRITE LOW-LEVEL SPECIFICATIONS

BDD doesn't stop at the acceptance tests. BDD also helps developers write higher quality code that's more reliable, more maintainable, and better documented.

Developers practicing BDD typically use an *outside-in* approach. When they implement a feature, they start from the acceptance criteria and work down, building whatever is needed to make those acceptance criteria pass. The acceptance criteria define the expected outcomes, and the developer's job is to write the code that produces those outcomes. This is a very efficient, focused way of working. Just as no feature is implemented unless it contributes to an identified business goal, no code is written unless it contributes to making an acceptance test pass, and therefore to implementing a feature.

But it doesn't stop there. Before writing any code, a BDD developer will reason about what this code should actually do and express this in the form of a *low-level executable specification*. The developer won't think in terms of writing unit tests for a particular class, but of writing technical specifications describing how the application should behave, such as how it should respond to certain inputs or what it should do in a given situation. These low-level specifications flow naturally from the high-level acceptance criteria, and help developers design and document the application code in the context of delivering high-level features (see figure 1.9).

For example, the step definition code in figure 1.9 involves creating a new account:

```
@Given("my $accountType account has a balance of $amount")
public void setupInitialAccount(AccountType type, double amount) {
    Account account = Account.ofType(type)
                             .withInitialBalance(amount);
    ...
}
```

Create a new account of given type and with given initial balance.

This leads the developer to write a low-level specification to design the `Account` class. This example uses Spock, a BDD unit testing library for Java and Groovy. The corresponding specification takes the following form:

```
class WhenCreatingANewAccount extends Specification {

    def "account should have a type and an initial balance" () {
        when:
            Account account = Account.ofType(Savings)
                                     .withInitialBalance(100)
        then:
            account.accountType == Savings
            account.balance == 100
    }
}
```

Figure 1.9 Low-level specifications, written as unit tests, flow naturally from the high-level specifications.

You could also write this specification using conventional unit-testing tools, such as JUnit or NUnit, or more specialized BDD tools such as RSpec (see figure 1.10).

Executable specifications like this are similar to conventional unit tests, but they're written in a way that both communicates the intent of the code and provides a worked example of how the code should be used. Writing low-level executable specifications this way is a little like writing detailed design documentation, with lots of examples, but using a tool that's easy and even fun for developers.

Unit testing the space shuttle

Good developers have known the importance of unit testing for a very long time. The IBM Federal Systems Division team was fully aware of their importance when they wrote the central avionics software for NASA's space shuttle in the late seventies. The approach they took to unit testing, where the unit tests were designed using the requirements and with examples provided by the business, has a surprisingly modern feel to it:

(continued)

"During the development activity, specific testing was done to ensure that the mathematical equations and logic paths provided the results expected. These algorithms and logic paths were checked for accuracy and, where possible, compared against results from external sources and against the system design specification (SDS)."[18]

At a more technical level, this approach encourages a clean, modular design with well-defined interactions (or APIs, if you prefer a more technical term) between the modules. It also results in code that's reliable, accurate, and extremely well tested.

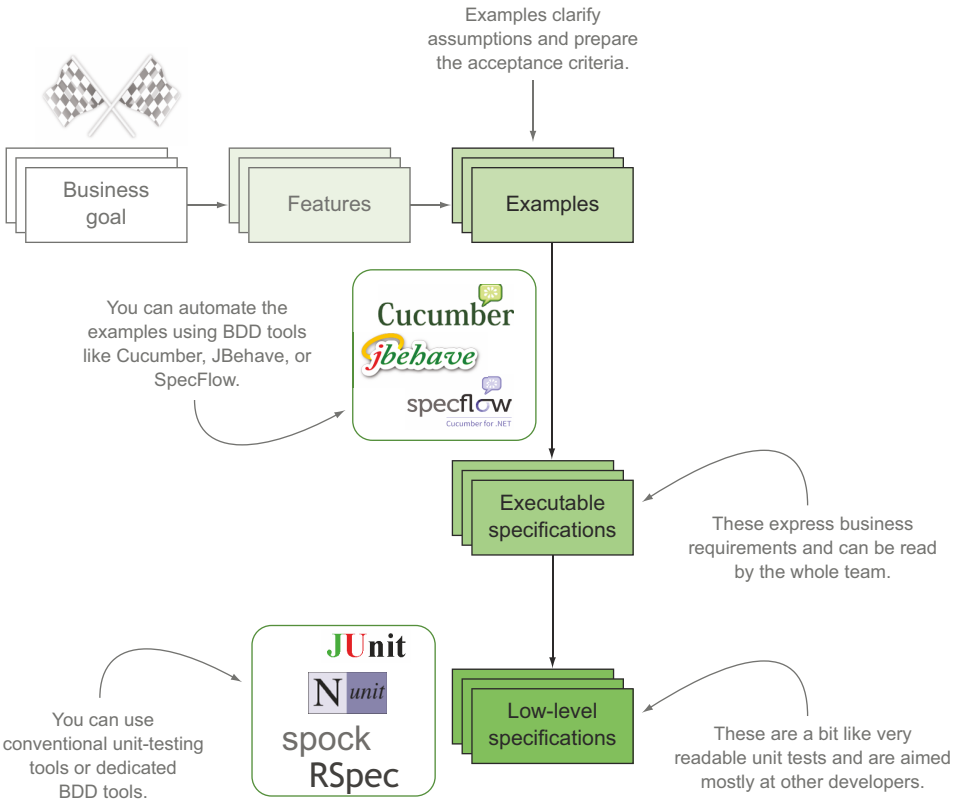

Figure 1.10 High-level and low-level executable specifications are typically implemented using different tool sets.

[18] William A. Madden and Kyle Y. Rone, "Design, Development, Integration: Space Shuttle Primary Flight Software System," *Communications of the ACM* (September 1984).

DELIVER LIVING DOCUMENTATION

The reports produced by executable specifications aren't simply technical reports for developers but effectively become a form of product documentation for the whole team, expressed in a vocabulary familiar to users (see figure 1.11). This documentation is always up to date and requires little or no manual maintenance. It's automatically produced from the latest version of the application. Each application feature is described in readable terms and is illustrated by a few key examples. For web applications, this sort of living documentation often also includes screenshots of the application for each feature.

Experienced teams organize this documentation so that it's easy to read and easy for everyone involved in the project to use (see figure 1.12). Developers can consult it to see how existing features work. Testers and business analysts can see how the features they specified have been implemented. Product owners and project managers can use summary views to judge the current state of the project, view progress, and decide what features can be released into production. Users can even use it to see what the application can do and how it works.

Just as automated acceptance criteria provide great documentation for the whole team, low-level executable specifications also provide excellent technical documentation for other developers. This documentation is always up to date, is cheap to maintain, contains working code samples, and expresses the intent behind each specification.

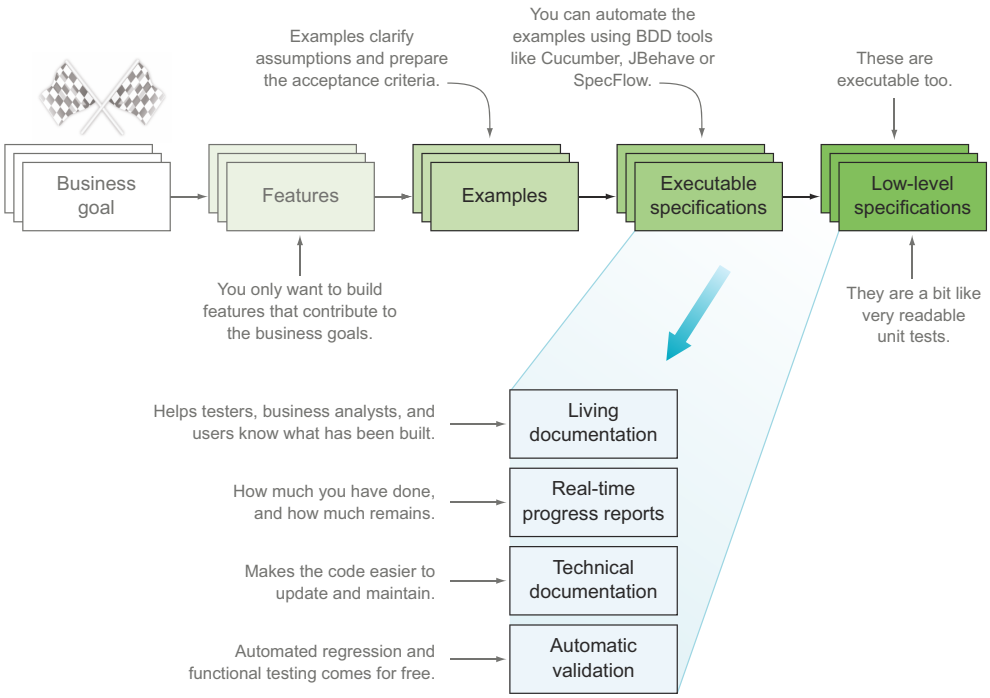

Figure 1.11 High-level and low-level executable specifications generate different sorts of living documentation for the system.

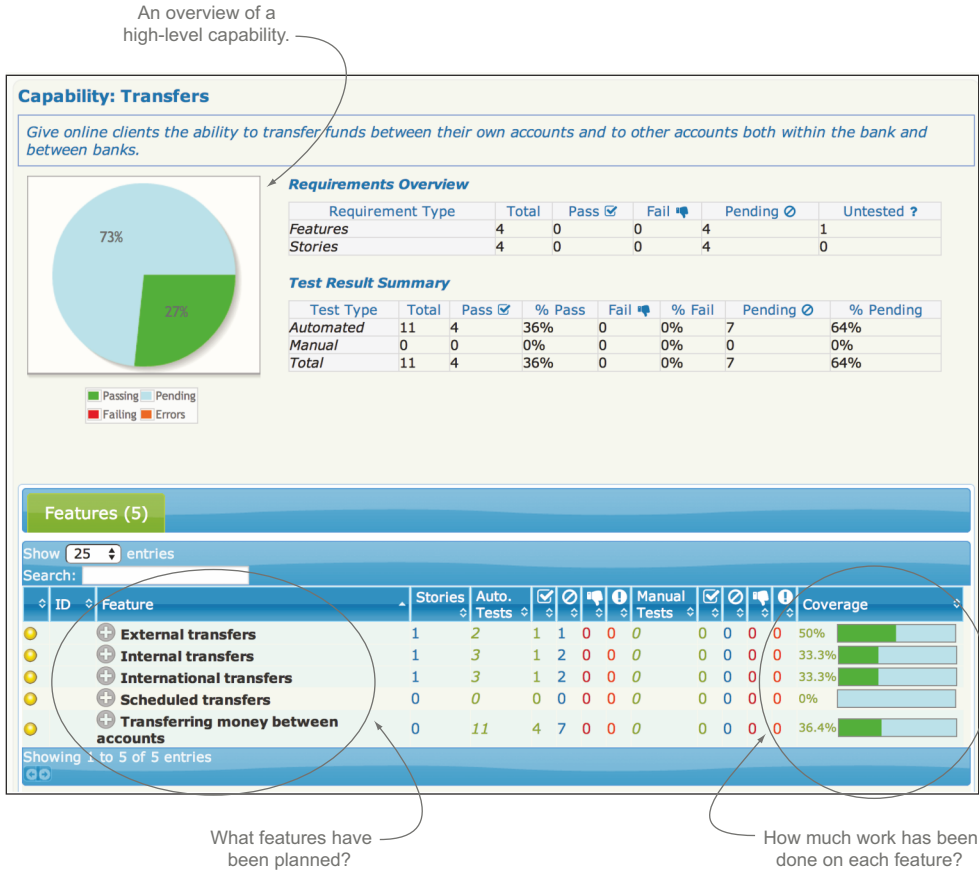

Figure 1.12 Well-organized living documentation can give an overview of the state of a project, as well as describe features in detail.

USE LIVING DOCUMENTATION TO SUPPORT ONGOING MAINTENANCE WORK

The benefits of living documentation and executable specifications don't stop at the end of the project. A project developed using these practices is also significantly easier and less expensive to maintain.

According to Robert L. Glass (quoting other sources), maintenance represents between 40% and 80% of software costs. Although many teams find that the number of defects drops dramatically when they adopt techniques like BDD, defects can still happen. Ongoing enhancements are also a natural part of any software application.[19]

In many organizations, when a project goes into production, it's handed over to a different team for maintenance work. The developers involved in this maintenance work have often not been involved in the project's development and need to learn the

[19] Robert L. Glass, *Facts and Fallacies of Software Engineering* (Addison-Wesley Professional, 2002).

code base from scratch. Useful, relevant, and up-to-date functional and technical documentation makes this task a great deal easier.

The automated documentation that comes out of a BDD development process can go a long way toward providing the sort of documentation maintenance teams need in order to be effective. The high-level executable specifications help new developers understand the business goals and flow of the application. Executable specifications at the unit-testing level provide detailed worked examples of how particular features have been implemented.

Maintenance developers working on a BDD project find it easier to know where to start when they need to make a change. Good executable specifications provide a wealth of examples of how to test the application correctly, and maintenance changes will generally involve writing a new executable specification along similar lines or modifying an existing one.

The impact of maintenance changes on existing code is also easier to assess. When a developer makes a change, it may cause existing executable specifications to break, and when this happens, there are usually two possible causes:

- The broken executable specification may no longer reflect the new business requirements. In this case, the executable specification can be updated or (if it's no longer relevant) deleted.
- The code change has broken an existing requirement. This is a bug in the new code that needs to be fixed.

Executable specifications are not a magical solution to the traditional problems of technical documentation. They aren't guaranteed to always be meaningful or relevant—this requires practice and discipline. Other technical, architectural, and functional documentation is often required to complete the picture. But when they're written and organized well, executable specifications provide significant advantages over conventional approaches.

1.4 Benefits of BDD

In the previous sections, we examined what BDD looks like and discussed what it brings to the table. Now let's run through some of the key business benefits that an organization adopting BDD can expect in more detail.

1.4.1 Reduced waste

BDD is all about focusing the development effort on discovering and delivering the features that will provide business value, and avoiding those that don't. When a team builds a feature that's not aligned with the business goals underlying the project, the effort is wasted for the business. Similarly, when a team writes a feature that the business needs, but in a way that's not useful to the business, the team will need to rework the feature to fit the bill, resulting in more waste. BDD helps avoid this sort of wasted effort by helping teams focus on features that are aligned with business goals.

BDD also reduces wasted effort by enabling faster, more useful feedback to users. This helps teams make changes sooner rather than later.

1.4.2 Reduced costs

The direct consequence of this reduced waste is to reduce costs. By focusing on building features with demonstrable business value (building the right software), and not wasting effort on features of little value, you can reduce the cost of delivering a viable product to your users. And by improving the quality of the application code (building the software right), you reduce the number of bugs, and therefore the cost of fixing these bugs, as well as the cost associated with the delays these bugs would cause.

1.4.3 Easier and safer changes

BDD makes it considerably easier to change and extend your applications. Living documentation is generated from the executable specifications using terms that stakeholders are familiar with. This makes it much easier for stakeholders to understand what the application actually does. The low-level executable specifications also act as technical documentation for developers, making it easier for them to understand the existing code base and to make their own changes.

Last, but certainly not least, BDD practices produce a comprehensive set of automated acceptance and unit tests, which reduces the risk of regressions caused by any new changes to the application.

1.4.4 Faster releases

These comprehensive automated tests also speed up the release cycle considerably. Testers are no longer required to carry out long manual testing sessions before each new release. Instead, they can use the automated acceptance tests as a starting point, and spend their time more productively and efficiently on exploratory tests and other nontrivial manual tests.

1.5 Disadvantages and potential challenges of BDD

While its benefits are significant, introducing BDD into an organization isn't always without its difficulties. In this section, we'll look at a few situations where introducing BDD can be more of a challenge.

1.5.1 BDD requires high business engagement and collaboration

BDD practices are based on conversation and feedback. Indeed, these conversations drive and build the team's understanding of the requirements and of how they can deliver business value based on these requirements. If stakeholders are unwilling or unable to engage in conversations and collaboration, or they wait until the end of the project before giving any feedback, it will be hard to draw the full benefits of BDD.

1.5.2 BDD works best in an Agile or iterative context

BDD requirements-analysis practices assume that it's difficult, if not impossible, to define the requirements completely upfront, and that these will evolve as the team (and the stakeholders) learn more about the project. This approach is naturally more in line with an Agile or iterative project methodology.

1.5.3 BDD doesn't work well in a silo

In many larger organizations, a siloed development approach is still the norm. Detailed specifications are written by business analysts and then handed off to development teams that are often offsite or offshore. Similarly, testing is delegated to another, totally separate, QA team. In organizations like this, it's still possible to practice BDD at a coding level, and development teams will still be able to expect significant increases in code quality, better design, more maintainable code, and fewer defects. But the lack of interaction between the business analyst teams and the developers will make it harder to use BDD practices to progressively clarify and understand the real requirements.

Similarly, siloed testing teams can be a challenge. If the QA team waits until the end of the project to intervene, or does so in isolation, they'll miss their chance to contribute to requirements earlier on, which results in wasted effort spent fixing issues that could have been found earlier and fixed more easily. Automating the acceptance criteria is also much more beneficial if the QA team participates in defining, and possibly automating, the scenarios.

1.5.4 Poorly written tests can lead to higher test-maintenance costs

Creating automated acceptance tests, particularly for complex web applications, requires a certain skill, and many teams starting to use BDD find this a significant challenge. Indeed, if the tests aren't carefully designed, with the right levels of abstraction and expressiveness, they run the risk of being fragile. And if there are a large number of poorly written tests, they'll certainly be hard to maintain. Plenty of organizations have successfully implemented automated acceptance tests for complex web applications, but it takes know-how and experience to get it right. We'll look at techniques for doing this later on in the book.

1.6 Summary

In this chapter you were introduced to Behavior-Driven Development. Among other things, you learned the following:

- Successful projects need to build software that's reliable and bug-free and to build features that deliver real value to the business.
- BDD practitioners use conversations about concrete examples to build up a common understanding of what features will deliver real value to the organization.
- These examples form the basis of the acceptance criteria that developers use to determine when a feature is done.

- Acceptance criteria can be automated using tools like Cucumber, JBehave, or SpecFlow to produce both automated regression tests and reports that accurately describe the application features and their implementation.
- BDD practitioners implement features with a top-down approach, using the acceptance criteria as goals, and describing the behavior of each component with unit tests written in the form of executable specifications.
- The main benefits of BDD include focusing efforts on delivering valuable features, reducing wasted effort and costs, making it easier and safer to make changes, and accelerating the release process.

In the next chapter, we'll take a flying tour of what BDD looks like in the flesh, all the way from requirements analysis to automated unit and acceptance tests and functional test coverage reports. So without further ado, let's get started!

AGILE DEVELOPMENT/TESTING

BDD IN ACTION

John Ferguson Smart

You can't write good software if you don't understand what it's supposed to do. Behavior Driven Development (BDD) encourages teams to use conversation and concrete examples to build up a shared understanding of how an application should work and which features really matter. With an emerging body of best practices and sophisticated new tools that assist in requirement analysis and test automation, BDD has become a hot, mainstream practice.

BDD in Action teaches you BDD principles and practices and shows you how to integrate them into your existing development process, no matter what language you use. First, you'll apply BDD to requirements analysis so you can focus your development efforts on underlying business goals. Then, you'll discover how to automate acceptance criteria and use tests to guide and report on the development process. Along the way, you'll apply BDD principles at the coding level to write more maintainable and better documented code.

What's Inside

- BDD theory and practice
- How BDD will affect your team
- BDD for acceptance, integration, and unit testing
- Examples in Java, .NET, JavaScript, and more
- Reporting and living documentation

No prior experience with BDD is required.

John Ferguson Smart is a specialist in BDD, automated testing, and software lifecycle development optimization.

"Delivers a thorough treatment of the current state of BDD tools."
—From the Foreword by Dan North, Creator of BDD

"Learn BDD from top to bottom."
—Dror Helper, CodeValue

"The first complete step-by-step guide to BDD."
—Marc Bluemner
liquidlabs GmbH

"Many useful techniques, tools, and concepts to make you more productive."
—Karl Métivier
Facilité Informatique

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/BDDinAction

ISBN 13: 978-1-617291-65-4
ISBN 10: 1-617291-65-X
54999
9 781617 291654

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]