Learning Yii Testing

Testing your application often and early is a key consideration in modern web development. PHP developers using the high-performance PHP framework Yii have access to Codeception, a testing suite that provides comprehensive support for unit, functional, and acceptance tests. With Codeception, you will quickly learn how to create fast, stable applications using a test-driven development methodology. Learning Yii Testing embraces a testing-first strategy to guide you in the implementation of all possible types of tests, covering the core features of Yii 2. Starting with the nuts and bolts of testing, you'll walk through Yii 2's new functionality and improvements and learn how to begin testing with it. You'll explore the world of Codeception and gain insights into key topics such as testing in isolation and component integration tests.

You'll also learn to analyze testing information and improve the quality of your code. At the end of the book, you will be able to use automation tools to reduce your testing workload.

Who this book is written for

Developers who have a solid pre-existing knowledge of Yii's core concepts will find this book an ideal introduction to learning to write tests using Yii 2's tools. You'll learn to create faster and more reliable applications with less time and effort.

What you will learn from this book

- Understand what testing is, why you need it, and what it encompasses
- Explore testing using all the features and improvements made available with Yii 2
- Embrace the Codeception testing framework and exploit its potential
- Carefully manage the scope of tests according to the application you're writing
- Write PHPUnit tests using the full range of its capabilities
- Script functional tests and test REST interfaces
- Write acceptance tests and make the best use of existing frameworks such as Selenium
- Discover how to abstract the knowledge you've learned and apply it to any situation


Matteo Pescarin
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'The Testing Mindset'
- A synopsis of the book’s content
- More information on Learning Yii Testing

About the Author

Matteo Pescarin started his career as a filmsetter. He cofounded a digital agency in 1998 with Emanuele Tozzato and became an expert in Flash in 2001, giving talks and writing guides on it. He later decided to distance himself from closed source technologies and committed himself to open standards and open source, joining the Gentoo Channel Italia, an Italian-wide Linux User Group dedicated to the famous distro. He became passionate about HTML and XML-related semantic technologies, combining, once again, his passion for 2D graphics with programming.

Matteo moved to London, UK, in 2009 after getting a CS degree. While still working on LAMP technologies, he started learning and specializing on the project management and quality assurance side of work, where his major interests still lie.

During his spare time, Matteo pretends to be a world-renowned artist and a formidable cook.
Learning Yii Testing

Since I've stumbled upon Yii first and Codeception later on, I couldn't believe that someone actually thought of solving the eternal damnation of a difficult testing panorama that most web developers, including me, have suffered with for so many years.

I've poured a good part of myself into this book, hoping that I could finally create a book that I never managed to find when I needed to learn about testing.

I believe the hardest part has been concentrating all of the accumulated experience, reading, conferences, and chats I've had about quality assurance, testing, and project management.

But this wouldn't have been possible without the great effort that has been put into Yii in its current version (version 2) and Codeception. Both these pieces of software, and the rest that are noted throughout the book, are the result of the efforts of hundreds of developers across the globe.

With all of this, together with the compassion and patience of Cristina, who tolerated my restless evenings to exhaustion, I have the pleasure to release this book, hoping that you will find an inspiration to discuss, improve, and contribute to the testing and web development community.
What This Book Covers

Chapter 1, The Testing Mindset, starts by defining the concepts used throughout the book, tries to explain why testing is so important, introduces the major testing techniques, and shows you how to get into the right mindset to approach this book.

Chapter 2, Tooling up for Testing, introduces Yii 2 with an overview for you to understand how the code is organized. We also start defining the work that will be carried over to the remaining chapters from the testing perspective.

Chapter 3, Entering Codeception, introduces Codeception and explains what it does, how it's structured, and how it works.

Chapter 4, Isolated Component Testing with PHPUnit, demonstrates PHPUnit. The first unit tests are implemented in this chapter, with the help of data providers.

Chapter 5, Summoning the Test Doubles, introduces test doubles with the use of mocks and stubs, while still sticking to PHPUnit. We also appreciate an alternative BDD-like syntax to write our tests.

Chapter 6, Testing the API – PHPBrowser to the Rescue, gives an overview of functional tests and then shows their expansion relative to the REST interface that Yii 2 allows you to create.

Chapter 7, Having Fun Doing Browser Testing, finally shows some live action with acceptance tests using Selenium WebDriver.

Chapter 8, Analyzing Testing Information, covers some more advanced topics about tests and code optimization techniques, thanks to the reports generated by Codeception and other tools.

Chapter 9, Eliminating Stress with the Help of Automation, is a more advanced chapter and introduces continuous integration with the aim of automating tests and displaying their reports using Jenkins CI.
The Testing Mindset

This book has been written with the intention of teaching how to use Codeception in conjunction with Yii 2. By using these two great frameworks, I confirmed that testing could finally become something that anyone would appreciate doing, rather than considering it an odd and not particularly clear appendix of development.

For this reason, this first chapter tries to address several aspects that are rarely touched on but hopefully should give you the understanding and required push to learn and adopt testing, and, on a larger scale, promote testing as a way to improve development as a methodology.

In this chapter, you will see the reason for testing and why testing should be planned into a project, and not done as an afterthought.

You will also see what will happen when you start testing: the implicit and explicit benefits for the short and long term, such as a change of mentality toward testing, the ability to improve component specifications, and architectural, design, and implementation choices, as well as refactoring, redistribution, and overall quality of code.

In order to explain why testing is so important, I'll also briefly dive into the organizational part of the process where Test Driven Development (TDD) and Behavior Driven Development (BDD) will be explained in relation to modern project management techniques, such as Agile and XP in a multi-skilled, self-organized team.

You will also see how the whole team environment can be improved and re-organized to help share knowledge and speed up workflow.
This chapter has been divided into the following three sections:

- Understanding the importance of testing
- Involving project management
- Obtaining the testing mindset

**Understanding the importance of testing**

Since I started getting into quality assurance and testing in a professional way, I've never faced myself with the question of "what is testing?"

I have to be honest, but during my time at university, testing wasn't part of any course. I don't really know if this has changed recently, nor if what's being taught is of any importance or relevance to the business world.

In this book, I've tried to combine the practicality of development and testing using a great PHP framework, Yii at its version 2, and the testing suite of Codeception. I will present each topic with a keen eye on the actual benefit for the team, while showing from a higher perspective the planning and the organization of the work. Throughout the chapters in this book, I'll shift back and forth, trying to give a clear understanding of the details you will be working on and the scope of the work, the overall aim from a testing perspective.

But, before we venture into this journey, what effectively is testing? Google engineering director James Whittaker's words make a very good answer to that:

"Although it is true that quality cannot be tested in, it is equally evident that without testing, it is impossible to develop anything of quality."

There are aspects of testing that are so completely fused with development that the two end up being practically indistinguishable from each other, and at other times it is so independent that developers are not even aware that this is happening.

In the whole project lifespan, you start with ideas and transform them into features or stories, breaking them down into tasks. From there you move into the execution of each of these tasks that will hopefully grant you, at the end of them all, a finished product.

At any point in our development process, we have tried to put some level of quality in it, either by "checking" the page loaded or by doing some more smart and deep, if not automated, testing.
Atlassian’s QA team lead Penny Wyatt points out that teams where quality assurance was not performed or left alone to perform small automation tasks, with unit tests for instance, had the highest story rejection rate, which is when a story will have to be re-opened after being completed because of wrong or missing functionality. We are talking about a 100 percent of rejection rate.

When such a situation occurs, we are left in a state where we have to go back into development and fix what we’ve done. This is not the only case: together with it, late discovered bugs and defects, and fixing them, are possibly some of the most expensive tasks in software development. In most cases, it has been shown that their cost is also way higher than it would have been to prevent them in the first place. Unfortunately, software development is rarely devoid of defects, and this should always be kept in mind.

As developers and managers, one of the goals we should have is to reduce the occurrence of defects to an economically acceptable level, and doing this also reduces the resulting risk associated with it.

As a practical example, a large website might have thousands of software errors but still be economically viable due to the fact that 99 percent of the website is displayed correctly. For a Falcon rocket or a Formula 1 car, a defect rate that high is not acceptable: the risk of having a single one in the wrong place might also cost the lives of people.

The other implicit aim for defect reduction is an investment in teamwork. An error introduced by one developer can have a ripple effect on the work of other team members and, overall, trust in the code base and other colleagues’ work. In this chapter, and the later chapters, we are going to discuss this aspect in more detail by introducing some concepts of project management and how it can cooperate in ensuring that quality is ensured on many levels.

The last and possibly an equally important aspect is how testing can be used to document the code by example. This is rarely discussed or brought to the attention of developers, but we will see how tests can describe the functionality of our implementations in a way more precise manner than what PHP documentation comments can provide. I’m not saying documentation comments are useless, quite the contrary: in modern integrated development environments (IDE) such as NetBeans or PHPStorm, auto-completion and code hinting are a great way to improve the time to discover the underlying framework without having the need to search through a reference manual. Tests can and should in fact provide the much needed help a developer might need when trying to combine and use yet unknown interfaces.
When working with open source software that is a result of the work of a small self-organized team, having the ability to provide documentation without an extensive effort might be the key to rapid and continuous delivery.

But how do we ensure a delivery can be met within constraints that are imposed on the team? In order to explain this, we will have to take a quick detour into project management, from where some of the practices that are discussed and used in this book have been originating.

Involving project management

If you ever have been involved in the planning phase of software development, or if you've worked as a project manager, you should have well in mind that there are three basic variables that you can leverage upon in order to manage projects:

- Time
- Quality
- Cost

In most of the business scenarios described theoretically and practically, the stakeholders decide to fix two of these variables, leaving the team to estimate the third. In other words:

*Time, Quality and Cost... pick two.*

In reality, what normally happens is that time and cost end up being set outside the project, thus leaving quality as the only variable developers can play on.

You might have already experienced that lowering quality doesn't eliminate work, it will just postpone it. Bringing back what we said earlier regarding defect rates, reducing quality might actually end up increasing the costs in the long run, leaving technical debt to spiral out of control if not causing a lot of problems in the short term.

The term **technical debt** has been introduced as a metaphor referring to the consequences of bad design, or architectural or development choices in a codebase. A number of books have been written specifically to counterbalance bad practices that are not naturally aimed at managing it.

One of the agile methodologies that are nowadays not particularly talked about, **Extreme Programming (XP)**, has introduced, if not rather exposed, a new variable into the equation: **scope**.
By making scope explicit, it does the following:

- Creates a safe way to adapt
- Provides a way to negotiate
- Gives us a tool to keep requests and demands under control

From the XP point of view, after the breakdown phase, we will have to go through a phase of estimating each single task, and based on the budget, you just keep adding or removing tasks.

This discussion brings up a problem that is currently widely discussed in the community, as estimating tasks is not as easy one might think. We’ll dive into it shortly, as I’ve seen too many misunderstandings of this topic.

Estimating tasks

As we’ve seen, estimation of the tasks has always been considered one of the fundamental principles of how the delivery path of a project is scheduled. This is especially valid in agile methodologies, as they use fixed time iterations and compute the amount of features and tasks that can be fitted in the given sprint and adjusted at each iteration using tools such as the burn down chart.

If you’ve worked in agile environments, this should be pretty much easy to understand, and if you haven’t, then there’s plenty of information that can be gained by reading books or articles on SCRUM that are freely available online.

Unfortunately, with all the importance estimation has, it seems like nobody really looked deeper into it: there are plenty of articles that warn how much software development task estimations are always off by a factor of 2 or 3. So, should we just swallow the fact that we won’t get better at estimating or is there something more to it?

The "estimations do not work" argument is probably not correct either, and recently the hashtag #NoEstimates has sparked a bit of discussion online, which is probably worth including here.

As a matter of fact, estimations do work. The only detail that is normally overlooked is that the estimation is nearing the actual time spent on it depending on how much knowledge the developer has and how much controllable the environment is.
In fact, the reality is twofold: from one side, we will get better at estimating the greater our experience is, and, on the other side, our estimation will be closer to reality if there are less unknowns in our path.

This is well-known in project management as the Cone of Uncertainty.

What we really need to do is admit and expose all the aspects that would increase the risk and the uncertainty of our estimations, while trying to isolate what we know is going to take a specific amount of time.

As an example, having a fixed time investigation period to create working prototypes of the features we are going to implement sets a precedent for future computations, while human factors will need to be factored in.

While estimations are particularly important from the business perspective of software development and project management, we won't be touching them again in this book. I'd rather focus on more practical aspects of the development work flow.

Testing approaches

Extreme Programming tries to stress the investment in defect reduction. In order to do so, it introduces two basic principles: double-checking and Defect Cost Increase (DCI).

Double-checking is software testing. We know how a particular feature should be working, which can be represented through a test. When implementing such a feature, we know in a quasi-deterministic way that what we've done is actually correct.

The other principle of DCI can be used to increase the cost-effectiveness of testing. What DCI states is the following:

"The sooner you find a defect, the cheaper it is to fix."

*Kent Beck*
To make this even more clearer with an example, if you find a defect after years of development, it could take a lot of time to investigate both what the code was originally meant to be doing and what was the context it was developed in the first place. If, instead, you find the defect the minute it's being implemented, the cost to fix it will be minimal. This, clearly, doesn't even take into consideration all the hidden costs and risks that a severe bug can cause to critical sections of our code base; think about security and privacy for instance.

Not only the longer we wait the more difficult the defects will be to be amended, but also their cost will increase and have the potential to leave many residual defects.

This means that by adhering to DCI, firstly, we need to have shorter feedback cycles so that we can continuously find as many defects as possible, and, secondly, we will have to adopt different practices that can help us keep the cost and the quality untouched as much as we can.

The idea of finding defects rapidly and often has been formalized as **Continuous Integration (CI)** and requires bringing automated testing into play to avoid the costs spiraling out of control. This practice has gained a lot of momentum outside XP and it's currently used widely in many organizations regardless of the project management methodology adopted. We will see how CI and automation can be introduced in our work flow and development in more detail in **Chapter 9, Eliminating Stress with the Help of Automation**.

These practices defy entirely the idea of working in a waterfall way, as shown in the following figure:

```
[Diagram showing development, bugfixing, and testing stages with release dates]
```

The delivery path in a waterfall work flow
In waterfall, we have a combination of factors that could impact the quality of the work we're doing: in most of the situations where this was the norm, the specifications are not set at the beginning nor frozen at any time. This means that it's very likely we might not produce what the business is asking.

In other words, you would begin testing only after development, which is way too late, as you can see from the preceding figure: you will be unable to actually catch any of the defects in time for the release date. Unfortunately, as much as waterfall might feel natural, its effectiveness has been disproved multiple times and I won't invest more time on this topic.

It's worth mentioning that the definition of "waterfall", although without using this term specifically, was formalized by Winston W. Royce in 1970, when describing a flawed and non-working model for software management.

Since the advent of agile methodologies, which XP is part of, there has been a great effort to bring testing as early as possible.

Remember that testing is as important as development, so it should be quite clear that we need to treat it as a first-class citizen.

One of the common situations you might find yourself in is that even if you start testing right at the beginning while the code base is being developed, it could potentially raise more issues than those that are needed or can be addressed. The resulting situation will still generate a good amount of problems and technical debt that won't fit within the delivery path, as you can see in the following figure:
The team's goal is to eliminate all post-development testing and shift testing resources to the beginning. If you have forms of testing such as stress or load testing that highlight defects at the end of the development, try to bring them into the development cycle. Try to run these tests continuously and automatically.

Transitioning into a work flow that has testing at the beginning brings to the surface two main problems: the accumulation of technical debt and the inherent problem that developers and testers are considered two separate entities. Don't forget that there will be still some testing that will happen after development and will clearly need to be performed by third parties, but, nonetheless, let's stress the fact that our efforts are to reduce it as much as we can.

As I'll constantly remind you, testing is not someone else's problem. Instead, with this book I'm aiming at giving the developers all the tools that can make him/her a tester first. There are different approaches to this problem, and we'll address them shortly at the end of this chapter, when talking about the testing mentality.

**Introducing Test Driven Development**

If you have ever developed with tests in mind, you might have appreciated that getting it right from the beginning is crucial. So, what do we need to test?

Throughout the years, various methodologies have been created that provide a set of rules for the developer that address how to include testing in the development cycle. The first and most well-known is **Test Driven Development**.

The main objective of adopting TDD as a practice in your team is to achieve the test-first mentality, and this is done using the concept of Red-Green-Refactor. You implement the tests first, which shouldn't pass (red status), you implement the interface being tested allowing the tests to pass (green status), and then you refactor the interface to improve what the test has highlighted, if needed.

We've seen the benefits from the management point of view of using this approach, but there's a more direct impact on the developer side. TDD in fact allows you to achieve what is being taught in software development, that interfaces shouldn't be influenced by the implementations. And, as a secondary effect, it provides, as we've seen, a way to document the interface itself.

By implementing tests first, you focus on how the method, class, and interface should be used by anyone inside or outside your team. This is called **black box testing**, which means that our tests should be completely unaware of the implementation details. This brings the implicit benefit of allowing the implementation to change over time.
If you’re interested in this topic, you might find worth exploring the Design by Contract (DbC) specification that allows you to describe interfaces in a more formal way in specific object-oriented programming languages. A good starting point might be found at http://c2.com/cgi/wiki?DesignByContract.

Unfortunately, TDD tries to focus on the atomic part of the features being developed, and it fails to give a broader vision of everything, of what has been tested and how much, or, even better, if what has been tested is of any relevance for the business and the product itself.

Once again, XP, in order to gain the full benefits of double-checking, introduces the following two sets of tests:

- One set written from the perspective of the programmers
- Another set written from the perspective of the users

In the first case, it allows the programmers to test all the system's components exhaustively, and in the latter case, the operation of the system as a whole.

The latter can in a way be seen as what Behavior Driven Development (BDD) describes in a more formal way. We’re going to cover BDD in more detail in Chapter 2, Tooling up for Testing.

BDD tries to cover TDD’s lack of overall scope and shifts the attention to the behavioral aspect of the project. BDD is effectively an evolution of TDD but requires some changes in the organization of the work and the way it’s shipped, which can be quite difficult to introduce in some environments without re-assessing the whole workflow.

With BDD, you define what to test and how to test it on multiple levels, detailing the scope of testing using a well-defined, business-oriented language called ubiquitous language, borrowed by Domain-driven Design (DDD) that is shared among all members of the team, both technical and non-technical. For the scope of this chapter, it should suffice to say that BDD introduces the concept of stories and scenarios giving the developer the ability to formally describe the user perspective and functionality of your application. Tests should be written using the standard agile framework of a user story: "As a [role] I want [feature] so that [benefit]." Acceptance criteria should be written in terms of scenarios and implemented as classes: "Given [initial context], when [event occurs], then [ensure some outcomes]."
Planning tests
Planning is, hence, critical when stepping into testing from a software development point of view, and in not-so-recent years, there have been several solutions to improve testing from a planning perspective that give a more detailed and compact way to define the so-called test plan.

In a testing-oriented environment, test plans should give you the direction and the indications of what and how much to test at any level. Moreover, the test plan is something that should be exposed to the various stakeholders and its visibility shouldn't live within the walls of development. Due to this, it's our responsibility to maintain and let this document live throughout the life of the project.

In practice, I've seen this rarely happening because test plans are never formalized or, if they are, they are too long and hard to maintain, suffering from a very short lifespan since their initial conception.

As an example, Attributes-Components-Capabilities (ACC) has been created by Google in order to solve some of the main problems that test plans have always suffered, especially their maintainability. You can find more information about ACC and Google Test Analytics software at https://code.google.com/p/test-analytics/.

ACC test plans are short and compact, and the whole project tries to aim to test plans that could be created in minutes and that are self-describing and valuable to anyone close to the project.

For each component, you have a series of capabilities, which can be described with one or more attributes; think, for instance, "secure", "fast", or "user-friendly". On top of this, each capability and component has a relative risk level associated with it. These two things together allow you to understand what is most important to test and how thorough your testing should be.

Generating tests
Clearly, planning tests is just the beginning. Once you get into the implementation side, you can pick up this book, which provides the knowledge of how to use the tools available to create tests.

There isn't much more I can tell you about this aspect. You probably just need to read it all, but it should be stressed that there are some basic principles you must keep in mind when writing tests.
Good tests exhibit the following three important characteristics:

- **Repeatability**: Tests must be deterministic. This ensures tests aren't dependent on external factors issues.
- **Simplicity**: Test only one thing. The smaller the test the more controllable it is.
- **Independence**: Tests should execute in isolation. There should be no dependency between tests. This also improves debugging of both the tests and your code.

Once you've got a grip of how to approach a project, viewing it from the architectural point of view, and once you've understood how test plans work and what you really need to test, you can start implementing tests, discuss them, and improve the tools and the way you're using them with the help of your colleagues.

**Obtaining the testing mindset**

So, up until now, we've seen how important testing is in current development practices and we've seen all kind of aspects that revolve around development itself from a project managing point of view, but still we don't know what's needed to become a good tester.

Finding developers knowledgeable about testing is particularly difficult, and there are a lot of talks online that address this problem: if that's so difficult, can't we do better? How do we get developers to become testers in the first place, especially when what you really want is to make developers responsible for the quality of the code they ship since the very beginning?

I tend to agree with the general idea that a tester, or a developer knowledgeable in testing, requires three basic things: **mindset**, **skillsets**, and **knowledge**.

So how do you get into acquiring or improving these three basic aspects?

Even if you can read all the books and listen to all the podcasts on testing, although these will give you a good amount of skillsets on how to test things and how the various testing suites and frameworks work, you won't be able to become a tester simply with that.

Of course, practice can help you a lot, but, all in all, the quality mindset and the knowledge of what to test are probably the hardest to acquire.
The knowledge part comes from a higher view of the product, both from the technical side and the business end. Introducing project breakdowns and pitches for the features that are going to be introduced in our software can be a starting point in this process.

The quality mindset can be the trickiest of them all, as it ends up being baked into all sort of aspects of software development from the technical point of view and requires a proactive participation from all the parties involved, first of all starting with the developer.

As previously said, there isn't a fixed definition of what you can achieve in terms of quality. There's no upper limit on how much quality you can put into your project. Hence, there's no limit on how much testing you can do in any project.

From what I've witnessed, there are two requirements that can speed up the process of becoming a good tester on top of being a good developer: one of these comes from the environment, the other comes from us.

The environment bit in my opinion is the one that could potentially cut the deal to acquire the right testing mindset that we are talking about, and getting there should probably be the priority of any company that decides that quality has value, and a measurable one.

Surely, having someone that can do mentoring on testing has always worked the best: learning by imitation and debating are probably the best team-oriented tools around. Even if you don't have a tester in your team, you might have noticed that in development, practices such as paired programming or code reviews can go a long way to keep the team up to speed with the practices and knowledge required.

Let's have a closer look at what this would mean in practice, keeping in mind that there is no silver bullet in terms of applied practices and methodologies, and it's your task to experiment and adapt based on what you have at hand.

**Starting with no testing culture – a practical approach**

In this series of practical examples on how to introduce testing in a team or project, we are going to assume a couple of requirements that are indispensable in getting you somewhere.

In this specific instance, we're going to assume you're working in a team. The ideal situation is to have a team of at least three people.
If you're working with less than three people, or you're a lone developer, most of the techniques and practices tend to have a cost which might be higher than the perceived benefit. A test plan and a sound organization of your workflow (trying to keep things simple) will not only provide a solid ground for working in a larger team if needed, but also grant you the instrument to deliver quality at speed.

First of all, you need support from the business and your direct managers; speaking of direct and indirect experience, without that you won't get anywhere. The business side of the company needs to understand what testing is, in the way that is has been described at the beginning of this chapter, the value of testing, and all the good things that this can bring. There is plenty of documentation online for you to build a business case.

Secondly, you need to have some skillsets in testing. This book should cover that part—hopefully quite well—and there are plenty of others that can teach you more theoretical aspects of testing for programmers and engineers, without considering the amount of online resources available on the topic.

A few good articles you can find online are as follows:

- Unit testing: Why bother? available at http://soundsoftware.ac.uk/unit-testing-why-bother/
- Testing at Airbnb available at http://nerds.airbnb.com/testing-at-airbnb/

Once you've got this, you can start moving into action.

One of the situations most might find themselves in is that there is no testing culture whatsoever. Here you have two choices: either take the bottom-up approach, and get yourself familiar with TDD as a starter, or take the top-down one, where you'll take the higher perspective.

Either way you need to start having a compact test plan to adhere to. Taking as an example the approach of ACC, you start by breaking down the application/project/library into modules (components), each of them will be composed of features (capabilities). Each feature will be denoted with a particular attribute. From there you should have a compact enough representation of what you're trying to achieve. On top of this, you can start assigning a relative risk level, which you will use to give priority for your testing approach defining what and how much to test.
The resulting test plan should be signed off by all the stakeholders and updated as frequently as possible, defining the aim of the project itself. The more this document is official the better it is, as it will be considered the business card of the project.

As highlighted by many, the immediate aim is to start planting the culture of testing in developers. Define the scope of your work, both in terms of testing and development, proceed with caution and evaluate both risks and costs, and leverage on those to take a decision on how to approach tests.

Thankfully, if you’re finding yourself working with Yii and Codeception, you should be spared a bit of headaches putting together different frameworks and wasting a bit of time experimenting a working solution.

Team-wise, when the experience in testing is not widespread nor solid, additional practices can be introduced that can help avoid bottlenecks or have all the knowledge trapped in a single person, such as paired programming and code reviews.

Some companies, like Atlassian, introduced test engineers that could help the teams, both from a mentoring perspective and a mere quality assurance side. Their interventions in the development cycle ended up being confined to a more restricted participation, at the very beginning and before completing the task. Their role is, nonetheless, fundamental, as they became the guardians of the testing infrastructure, the tools, and the practices to be adopted, while the developer grew to become a fully fledged tester who can cover almost any aspect of testing without much support.

**Summary**

In this chapter, we’ve covered many aspects that are directly connected with testing, but are not strictly necessary to start testing, although they are fundamental if you want to understand why you’ve taken up this book and if it’s necessary to go through the rest of it.

You’ve seen why it’s important to test, some project management methodologies, how to estimate tasks and what it entails, and you’ve seen different testing approaches such as TDD and BDD, which will be the basis for many of the remaining chapters. At the end, I’ve tried to give an idea of what it takes to gain the testing mindset required to become a master in this art.

In *Chapter 2, Tooling up for Testing*, we will start gearing up with the tools we are going to use throughout the rest of this book, understanding the basics of Yii 2 and applying what we’ve learned in this chapter by outlining our test plan.
Where to buy this book


Click here for ordering and shipping details.