Refactoring is a powerful tool, enabling you to improve the design of your software. It means taking your working code and changing it, sometimes deeply, to make it perform the same tasks it performed before changing it. Somehow this could sound quite pointless: why be at risk of losing a piece of working code just to improve its design? Shouldn’t the functionality of working code be valued more than its structure?

Any wise programmer should ask herself questions like these at the first mention of refactoring. We have already called attention many times to the importance of preserving our existing code base and why this should be our main goal. Then how can we change it without introducing errors? How should I explore new ways to arrange my software? Is it possible for me to refactor my code protected by a strong and soft safety net? Yes, it is, because automated tests are at hand, one of the most powerful tools ever in the history of software development.

Building Value One-Way

You have been coding thousands and thousands of lines of code for years, and you have spent thousands of hours debugging that code, but how often did you stop and think about how unproductive debugging your code is? Yes, we really mean it: debugging is not productive. Debugging is just a way to finally deliver what you were already supposed to have delivered.

Even the best of our customers is willing to pay to get some technical value to invest in her business. In a healthy environment the customer should also try to maximize her return on investment (ROI) by getting the most value for the lowest expense, taking into consideration any short-term goal together with longer-term ones. In the same healthy environment we should support these needs by trying to maximize our ROI by reducing costs, not by delivering lower quality.

Debugging is a step back from a healthy developer-customer system because debugging is always a cost. Even on a fixed price contract featuring a warranty option on bugged features, the customer will pay a cost for debugging, because a dollar lost now is not paid back by the same dollar next week. If debugging were considered a standard practice in the automotive market, we would be buying cars incapable of bringing us home 60–70% of the time, attached to warranties providing us with ones that actually function. It can happen, we know, but I bet you wouldn’t be pleased to see it happen 60–70% of the time. Those numbers are considered low in the software industry indeed!

Obviously this way of conceiving software development has its roots in the reality of the software industry and engineering, and no one should be faulted for debugging code if a defect is found. What we would like to advocate here is a way to model your software development process, being aware of the fact that during the last 20 years many techniques arose to reduce or even erase the need for debugging, making the development of software a one-way process, from customer requirements to implementation with few or no features bouncing between being done or in progress.
Chaos in a Cage

We have more than one kind of automatic test to rely upon. In this book we will focus on unit tests and functional tests. Unit tests are meant to be closer to the developer’s point of view, while functional tests are meant to test software correctness and conformity in fulfilling customer requirements from a user point of view. We will see those two kinds of tests in better detail in the following chapter, but we want to make clear here that both together constitute a way to defend your code from chaotic evolution and attack the complexity every developer inherently has to cope with.

Before we go on, let us introduce you to unit tests and functional tests.

Unit Tests

Unit tests confirm that a single unit of code computes the correct output when passed a well-defined input. The developer writes a test that automatically passes on a meaningful set of different inputs and checks whether the output is right. A unit test is meant to test only a single unit of code, thus any interaction between that code and some external actor providing a service or some data should be simulated in a safe way, so that a developer knows which few lines of code are wrong whenever a unit test fails. If anything like a database or a web service is needed by the code we are testing, it must be simulated with a fake version to avoid tainting our testing scope.

What does a unit test look like? It depends on the testing framework you use, but a wide range of frameworks sticks to the xUnit de facto standard. When writing this book, we opted for PHPUnit and, while we will describe its use in detail in the next chapter, we want you to have a first look right now. The following test is nothing more than a test class—yes, a class itself!—testing the Sale class to return the right price after a given state is set:

```php
class SaleTest extends PHPUnit_Framework_TestCase
{
    public function testGetPrice()
    {
        $sale = new Sale();
        $sale->amount = 10;
        $this->assertEquals(100, $sale->getPrice());
    }
}
```

With a complete suite of tests like that, a development team can prevent chaos from creeping into its design at a very low level, testing a single object’s behavior and—as too often overlooked—mutual interaction between objects.

Functional Tests

Functional tests are a powerful way to use our software on the end-user’s behalf and to verify its behavior’s correctness. They describe the interaction of the user with the machine we are building our software upon, reproducing every click and drag-and-drop relentlessly. This kind of test relies on the whole system to run with no isolation of units coming into play. Every testing action involves the user interface, the controller, the model, its logic, and real data as if it were a real person using the software we are testing.

What does a functional test look like? It depends a lot on the testing framework you decide to use. Unlike unit tests, at the time of writing this book, no shared standard has emerged yet, nor does some widely adopted technology seem to pave the way towards convergence to a common rule. For the purpose of explaining functional tests in this book we chose Selenium RC, a tool that is part of the