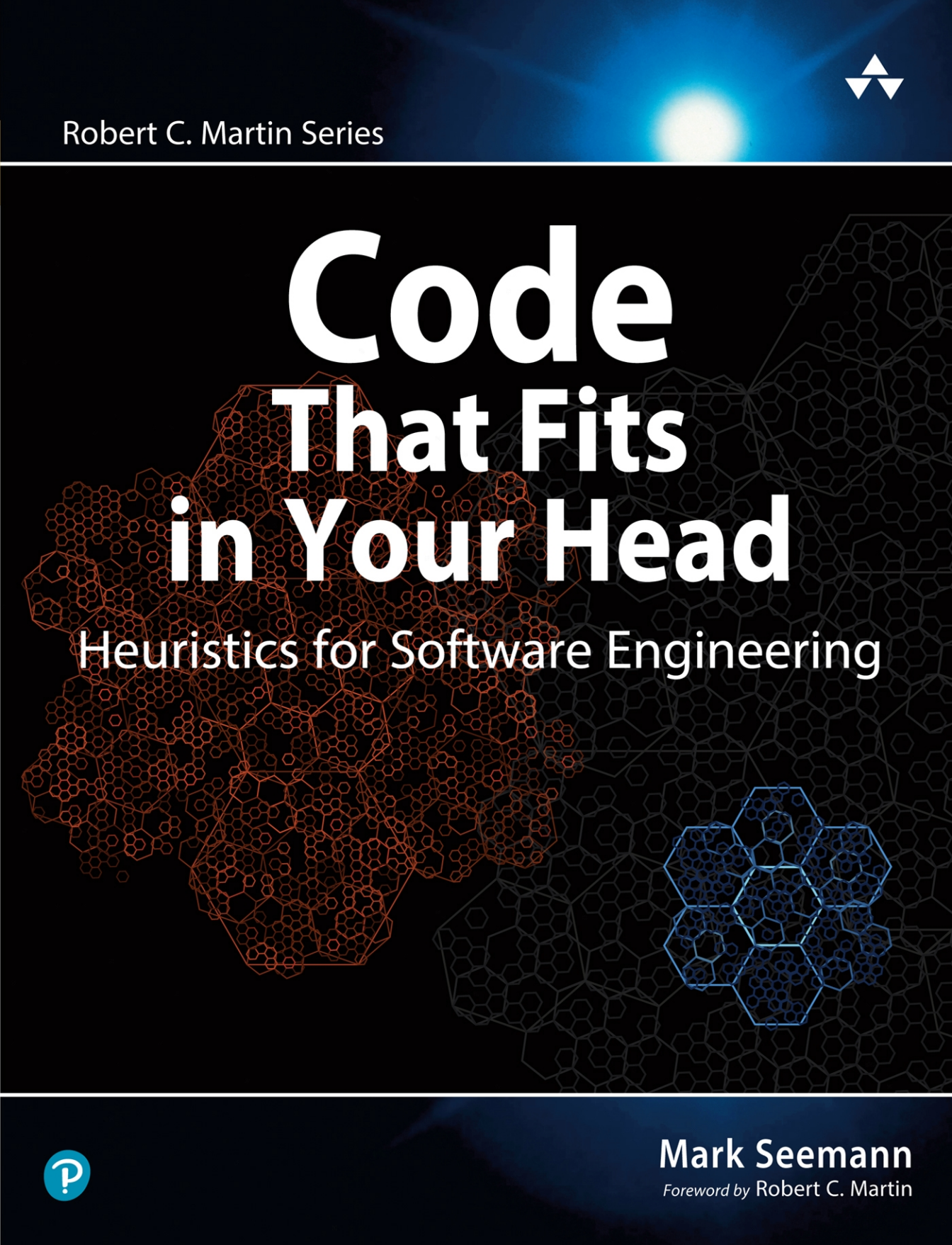
Robert C. Martin Series
Code
That Fits
in Your Head
Heuristics for Software Engineering
Mark Seemann
Foreword by Robert C. Martin

Praise for
Code That Fits in Your Head

"We progress in software by standing on the shoulders of those who came before us. Mark's vast experience ranges from philosophical and organisational considerations right down to the precise details of writing code. In this book, you're offered an opportunity to build on that experience. Use it."

—*Adam Ralph, speaker, tutor, and software simplifier, Particular Software*

"I've been reading Mark's blogs for years and he always manages to entertain while at the same time offering deep technical insights. *Code That Fits in Your Head* follows in that vein, offering a wealth of information to any software developer looking to take their skills to the next level."

—*Adam Tornhill, founder of CodeScene, author of* Software Design X-Rays *and* Your Code as a Crime Scene

"My favorite thing about this book is how it uses a single code base as a working example. Rather than having to download separate code samples, you get a single Git repository with the entire application. Its history is hand-crafted to show the evolution of the code alongside the concepts being explained in the book. As you read about a particular principle or technique, you'll find a direct reference to the commit that demonstrates it in practice. Of course, you're also free to navigate the history at your own leisure, stopping at any stage to inspect, debug, or even experiment with the code. I've never seen this level of interactivity in a book before, and it brings me special joy because it takes advantage of Git's unique design in a new constructive way."

—*Enrico Campidoglio, independent consultant, speaker and Pluralsight author*

"Mark Seemann not only has decades of experience architecting and building large software systems, but is also one of the foremost thinkers on how to scale and manage the complex relationship between such systems and the teams that build them."

—*Mike Hadlow, freelance software consultant and blogger*

This code base uses C#'s *nullable reference types* feature, and most of the `dto` properties are declared as nullable. Without the exclamation mark, the compiler complains that the code accesses a nullable value without checking for null. The `!` operator suppresses the compiler's complaints. With the exclamation marks, the code compiles.

This is a terrible hack. While the code compiles, it could easily cause a `NullReferenceException` at run time. Trading a compile-time error for a run-time exception is a poor trade-off. We should do something about that.

Another potential run-time exception lurking in listing 5.2 is that there's no guarantee that the `DateTime.Parse` method call succeeds. We should do something about that as well.

5.2 VALIDATION

With the code in listing 5.2, what happens if a client posts a JSON document without an `at` property?

You might think that `Post` would throw a `NullReferenceException`, but in reality, `DateTime.Parse` throws an `ArgumentNullException` instead. At least that method performs input validation. You should do the same.

How Is ArgumentNullException Better Than NullReferenceException?

Does it matter which exception is thrown by a method? After all, if you don't handle it, your program will crash.

Exception types seem to matter most if you can handle them. If you know that you can handle a particular type of exception, you can write a `try/catch` block. The problem is all the exceptions that you can't handle.

Typically, `NullReferenceException` happens when a required object is missing (null). If the object is required, but not available, there's not much that you can do about it. This is as true for `NullReferenceException` as it is for `ArgumentNullException`, so why bother to check for null only to throw an exception?

The difference is that a `NullReferenceException` carries no helpful information in its exception message. You're only told that some object was null, but not which one.

An `ArgumentNullException`, on the other hand, carries information about which argument was null.

If you encounter an exception message in a log or error report, which would you rather like to see? A `NullReferenceException` with no information, or an `ArgumentNullException` with the name of the argument that was null?

I'll take the `ArgumentNullException` any time, thank you.

The ASP.NET framework translates an unhandled exception to a `500 Internal Server Error` response. That's not what we want in this case.

5.2.1 Bad Dates

When input is invalid, an HTTP API should return `400 Bad Request` [2]. That's not what happens. Add a test that reproduces the problem.

Listing 5.3 shows how to test what happens when the reservation date and time is missing. You may wonder why I wrote it as a `[Theory]` with only a single test case. Why not a `[Fact]`?

I admit that I cheated a bit. Once again, the *art* of software engineering manifests itself. This is based on *the shifting sands of individual experience* [4] – I know that I'm going to add more test cases soon, so I find it easier to start with a `[Theory]`.

Listing 5.3 Test what happens when you post a reservation DTO with a missing `at` value.
(Restaurant/9e49134/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Theory]
[InlineData(null, "j@example.net", "Jay Xerxes", 1)]
public async Task PostInvalidReservation(
    string at,
    string email,
    string name,
    int quantity)
{
    var response =
        await PostReservation(new { at, email, name, quantity });
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}
```

The test fails because the response's status code is `500 Internal Server Error`.

You can easily pass the test with the code in listing 5.4. The major difference from listing 5.2 is the addition of the Null Guard.

Listing 5.4 Guard against null `At` property.
(Restaurant/9e49134/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
    if (dto.At is null)
        return new BadRequestResult();

    var r = new Reservation(
        DateTime.Parse(dto.At, CultureInfo.InvariantCulture),
        dto.Email!,
        dto.Name!,
        dto.Quantity);
    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

The C# compiler is clever enough to detect the Guard Clause, which means that you can remove the exclamation mark after `dto.At`.

You can add another test case where the `email` property is missing, but let's fast-forward one more step. Listing 5.5 contains two new test cases.

Listing 5.5 More test cases with invalid reservations.
(Restaurant/3fac4a3/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Theory]
[InlineData(null, "j@example.net", "Jay Xerxes", 1)]
[InlineData("not a date", "w@example.edu", "Wk Hd", 8)]
[InlineData("2023-11-30 20:01", null, "Thora", 19)]
public async Task PostInvalidReservation(
    string at,
    string email,
    string name,
    int quantity)
{
    var response =
        await PostReservation(new { at, email, name, quantity });
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}
```

The bottom `[InlineData]` attribute contains a test case with a missing `email` property, while the middle test case supplies an `at` value that's not a date and time.

Listing 5.6 Guard against various invalid input values.
(Restaurant/3fac4a3/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
    if (dto.At is null)
        return new BadRequestResult();
    if (!DateTime.TryParse(dto.At, out var d))
        return new BadRequestResult();
    if (dto.Email is null)
        return new BadRequestResult();

    var r = new Reservation(d, dto.Email, dto.Name!, dto.Quantity);
    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

Listing 5.6 passes all tests. Notice that I could remove another exclamation mark by guarding against a null email.

5.2.2 RED GREEN REFACTOR

Consider listing 5.6. It's grown in complexity since listing 4.15. Can you make it simpler?

This is an important question to regularly ask. In fact, you should ask it after each test iteration. It's part of the *Red Green Refactor* [9] cycle.

- **Red.** Write a failing test. Most test runners render a failing test in *red*.
- **Green.** Make as minimal change as possible to pass all tests. Test runners often render passing tests in *green*.
- **Refactor.** Improve the code without changing its behaviour.

Once you've moved through all three phases, you start over with a new failing test. Figure 5.2 illustrates the process.

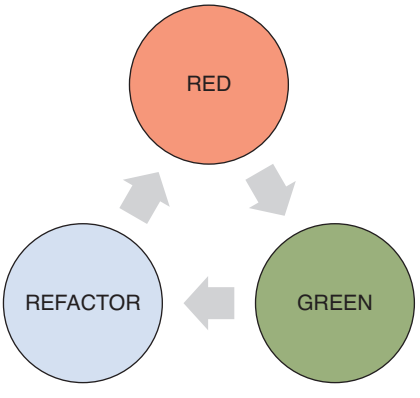

Figure 5.2 The Red Green Refactor cycle.

So far in the book's running example, you've only seen oscillations of red-green, red-green, and red-green. It's time to add the third phase.

The Science of Test-Driven Development

The Red Green Refactor process is one of the most scientific methodologies of software engineering that I can think of.

In the scientific method, you first form a hypothesis in the form of a prediction of a falsifiable outcome. Then you perform an experiment and measure the result. Finally, you compare the actual to the predicted outcome.

Does that sound familiar?

That sounds like the Arrange Act Assert [9] pattern, although we should be careful not to overextend the metaphor. The *act* phase is the experiment, and the *assert* phase is where you compare expected and actual outcomes.

The red and green phases in the Red Green Refactor cycle are small, ready-made science experiments in their own right.

In the red phase, the ready-made hypothesis is that when you run the test you just wrote, it should fail. This is a measurable experiment that you can perform. It has a quantitative outcome: it'll either pass or fail.

If you adopt Red Green Refactor as a consistent process, you may be surprised how often you get a passing test in this phase. Remember how easily the brain jumps to conclusions [51]. You'll inadvertently write tautological assertions [105]. Such false negatives happen, but you wouldn't discover them if you didn't perform the experiment.

Likewise, the green phase is a ready-made hypothesis. The prediction is that when you run the test, it'll succeed. Again, the experiment is to run the test, which has a quantifiable result.

(continues)