Automated Testing in R

Jakub Sobolewski

2025-01-01

Table of contents

Preface

Testing can feel like a chore.

You're coding away, everything seems fine—until it isn't. A bug pops up. You fix it, but then something else breaks. It's frustrating, time-consuming, and stressful. But what if it didn't have to be? What if testing wasn't just about catching bugs, but about making your work smoother, faster, and more reliable?

That's what this book is about.

R has grown from a tool for statisticians into a powerful programming language used for everything from exploratory data analysis to building Shiny apps and production-grade data pipelines. But as your projects grow, so do the risks. Code becomes harder to change. Bugs creep in. Shipping features slows down.

Testing can change that.

With automated testing, you can catch problems early, make changes with confidence, and keep your code in shape no matter how complex it gets. It's not just about preventing errors — it's about giving yourself the freedom to experiment and innovate without fear.

This book is your guide to testing in R.

We'll start with the fundamentals: why testing matters, what makes a good test, and how to think like a tester. From there, we'll get into the practical stuff. You'll learn how to write unit tests, handle tricky scenarios with mocks and stubs, and test Shiny apps. We'll cover advanced techniques, like snapshot testing and strategies for tackling legacy code.

But this isn't just about the "how." It's about the "why."

Testing isn't just a skill — it's a mindset. It's about approaching your work with care and confidence. It's about solving problems before they happen. And it's about making sure the code you write today won't break tomorrow.

Whether you're building packages, working on Shiny apps, or writing scripts for data analysis, this book will show you how testing can make your life easier — and your code better.

Let's get started.

The book assumes you have some experience with unit-testing and using {testthat}. If you are new to unit testing in R, I recommend you to read the {testthat} documentation first.

License

Part I

Testing fundamentals

1 Anatomy of a test

A good test makes it clear what it's checking.

When a test mixes setup, running the system, and checking results, it gets confusing. Separating these steps into clear phases helps make the test's purpose obvious. This way, anyone reading the test can follow along easily.

We can make this separation explicit with Arrange, Act, Assert comments.

Those comments aren't just for show. They help you structure your tests and keep them focused. They also make it easier to spot missing steps or unnecessary complexity. They keep your tests consistent, which makes them easier to read and maintain.

If you are starting to write a test, put those comments in place first, then fill in the blanks.

1.1 Arrange

The first part of each test should setup the environment for the tested code.

```
test_that("...", {
  # Arrange #<<
  machine <- deep_thought$connect() #<<
  question <- "What is the answer to life, the universe, and everything?" #<<

  # Act

  # Assert

})
```

1.2 Act

The second part of each test is to call the code that's being tested.

```
test_that("...", {
  # Arrange
  machine <- deep_thought$connect()
  question <- "What is the answer to life, the universe, and everything?"

  # Act #<<
  result <- ask_question(machine, question) #<<

  # Assert

})
```

1.3 Assert

The third part of each test is to assert that the code behaves as expected.

```
test_that("...", {
  # Arrange
  machine <- deep_thought$connect()
  question <- "What is the answer to life, the universe, and everything?"

  # Act
  result <- ask_question(machine, question)

  # Assert #<<
  expect_equal(result, 42) #<<
})
```

1.4 Teardown if needed

If we need to free resources, we should do that at the end of the test.

```
test_that(" ", {
  # Arrange
  machine <- deep_thought$connect()
  question <- "What is the answer to life, the universe, and everything?"

  # Act
  result <- ask_question(machine, question)
```

```
  # Assert
  expect_equal(result, 42)

  machine$disconnect() #<<
})
```

2 Types of tests

To make it easier to think about what to test and to make a more informed decision on how we need to test it, we may categorize tests into:

- Direct Response Tests.
- State Change Tests.
- Interaction Tests.

Let's see in what circumstances should each type be used.

2.1 Direct Response Tests

- They check whether a return value or an exception matches the expectation.
- These tests ensure that the core functionality of the code works correctly.

Example

```
describe("Stack", {
  it("should return the last pushed value when popping an item", {
    # Arrange
    my_stack <- Stack$new()
    my_stack$push(1)

    # Act
    value <- my_stack$pop()

    # Assert
    expect_equal(value, 1)
  })
})
```

Tips

- Don't test a lot of different values if the new combination doesn't test new behavior. Testing `mean(1:10)` and then `mean(1:100)` doesn't improve our confidence that `mean` function works as expected.
- Use assertions to convey intent. If you don't care about the order of a vector, consider using `testthat::expect_setequal` instead of `testthat::expect_equal` to only assert on its content.
- Don't duplicate assertions. If you already use `testthat::expect_equal` on a vector, does adding an assertion on its length with `testthat::expect_length` add more safety? Adding more assertions than needed will only make the test more difficult to change in the future.

2.2 State Change Tests

- These tests help validate the impact of certain actions on the system's state.
- They confirm that the behavior results in the expected changes, such as modifying a list and confirming its size change.

Example

```
describe("Stack", {
  it("should not be empty after pushing an item", {
    # Arrange
    my_stack <- Stack$new()

    # Act
    my_stack$push(1)

    # Assert
    expect_false(my_stack$empty())
  })
})
```

Tips

- Don't share state between tests. It may make tests more fragile and more difficult to understand.

- Avoid iteration. Don't check if `Stack` can handle 0, 1, 2, 3, 4, …, calls to `push`. Use chicken counting: **zero, one, or many.**

2.3 Interaction Tests

- These tests ensure proper communication and integration between different parts of the system.
- These tests examine how code interacts with external components, often simulating dependencies or external services. **Mocks, Fakes, Stubs and Dummies** are used to control these interactions and validate that the code interacts correctly with external entities.

Example

```
describe("Stack", {
  it("should log what item has been pushed", {
    # Arrange
    logger <- mockery::mock()
    my_stack <- Stack$new(logger)

    # Act
    my_stack$push(1)

    # Assert
    mockery::expect_args(
      logger,
      n = 1,
      "Pushed 1 onto the stack"
    )
  })
})
```

Tips

- Complex mock will make tests brittle and difficult to understand. They typically need to be created when interactions in the code are complex or not defined well enough.
- Notice how much setup is needed to run a test. Use this feedback to improve and simplify production code. Code that is easy to test is easier to maintain.

- Don't overdo them. There is a risk that you mock so many things that the test doesn't resemble the actual behavior of the system anymore.
- Mock public interfaces of objects, otherwise you risk spilling implementation details into the test.