6 TESTING

Writing, running, and fixing tests can feel like busywork. In fact, it's easy for tests to *be* busywork. Bad tests add developer overhead without providing value and can increase test instability. This chapter will teach you to test effectively. We'll discuss what tests are used for, different test types, different test tools, how to test responsibly, and how to deal with nondeterminism in tests.

The Many Uses of Tests

Most developers know the fundamental function of tests: tests check that code works. But tests serve other purposes as well: they encourage clean code, force developers to use their own APIs, document how components are to be interacted with, and serve as a playground for experimentation.

Above all, tests verify that software behaves as expected. Unpredictable behavior causes problems for users, developers, and operators. Initially, tests show that code works as specified. Tests then remain to shield existing behavior from new changes. When an old test fails, a decision must be made: did the developer intend to change behavior, or was a bug introduced?

Test writing also forces developers to think about the interface and implementation of their program. Developers usually first interact with their code in tests. New code will have rough edges; testing exposes clumsy interface design early so it can be corrected. Tests also expose messy implementation. Spaghetti code, or code that has too many dependencies, is difficult to test. Writing tests forces developers to keep their code well factored by improving separation of concerns and reducing tight coupling.

Code cleanliness side effects in tests are so strong that *test-driven development (TDD)* has become commonplace. TDD is the practice of writing tests before code. The tests fail when written, and then code is written to make them pass. TDD forces developers to think about behavior, interface design, and integration before cranking out a bunch of code.

Test serve as a form of documentation, illustrating how the code is meant to be interacted with. They are the first place an experienced programmer starts reading to understand a new codebase. Test suites are a great playground. Developers run tests with debuggers attached to step through code. As bugs are discovered, or questions about behavior arise, new tests can be added to understand them.

Types of Tests

There are dozens of different test types and testing methodologies. Our goal is not to cover the full breadth of this topic but to discuss the most common types—unit, integration, system, performance, and acceptance tests—to give you a firm foundation to build on.

Unit tests verify "units" of code—a single method or behavior. Unit tests should be fast, small, and focused. Speed is important because these tests run frequently—often on developer laptops. Small tests that focus on a single unit of code make it easier to understand what has broken when a test fails.

Integration tests verify that multiple components work together. If you find yourself instantiating multiple objects that interact with each other in a test, you're probably writing an integration test. Integration tests are often slower to execute and require a more elaborate setup than unit tests. Developers run integration tests less frequently, so the feedback loop is longer. These tests can flush out problems that are difficult to identify by testing standalone units individually.

IT'S ONLY OBVIOUS IN RETROSPECT

A few years ago, Dmitriy was shopping for a new dishwasher appliance. He read online reviews, went to a store, dutifully examined all the specs, considered the trade-offs, and finally settled on the model he liked best. The salesperson who insisted on guiding Dmitriy through the aisles checked the inventory, got ready to put in an order, and, just as his hand hovered over the ENTER key, paused. "Is this dishwasher going into a corner in your kitchen, by any chance?" "Why, yes, it is." "And is there a drawer that comes out of a cabinet at a 90-degree angle to where this dishwasher is going, such that it slides into the space right in front of the dishwasher door?" "Why, yes, there is such a drawer." "Ah," the salesperson said, removing his hand from the keyboard. "You will want a different dishwasher." The model Dmitriy selected had a handle that protruded from the door, which would have completely blocked the drawer from coming out. The perfectly functioning dishwasher and the perfectly functioning cabinet were completely incompatible. Clearly, the salesperson had seen this particular integration scenario fail before! (The solution was to purchase a similar model with an inset door handle.)

System tests verify a whole system. End-to-end (e2e, for short) workflows are run to simulate real user interactions in preproduction environments. Approaches to system test automation vary. Some organizations require that system tests pass before a release, which means all components are tested and released in lockstep. Other organizations ship such large systems that synchronizing releases is not realistic; these organizations often run extensive integration tests and supplement them with continuous synthetic monitoring production tests. *Synthetic monitoring* scripts run in production to simulate user registration, browse for and purchase an item, and so on. Synthetic monitoring requires instrumentation that allows billing, accounting, and other systems to distinguish these production tests from real activity. Synthetic monitoring is covered more in Chapter 9.

Performance tests, such as load and stress tests, measure system performance under different configurations. *Load tests* measure performance under various levels of load: for example, how a system performs when 10, 100, or 1,000 users access it concurrently. *Stress tests* push system load to the point of failure. Stress testing exposes how far a system is capable of going and what happens under excessive load. These tests are useful for capacity planning and defining service level objectives (see Chapter 9 for more details on these topics).

Acceptance tests are performed by a customer, or their proxy, to validate that the delivered software meets acceptance criteria. These tests are fairly common in enterprise software, where formal acceptance tests and criteria are laid out as part of an expensive contract. The International Standards Organization (ISO) requires acceptance tests that validate explicit business requirements as part of their security standard; certification auditors will ask for evidence of documentation for both the requirements and the corresponding tests. Less formal acceptance tests, found in less regulated organizations, are variations on the theme of, "I just changed a thing, can you let me know if everything still looks good?"

TESTING IN THE REAL WORLD

We looked at test setups of many successful open source projects while writing this chapter. Many projects were missing certain flavors of tests, while others were inconsistent about the separation—intermingling "unit" and "integration" tests. It's important to know what these categories mean, and the trade-offs between them. Still, don't get too wrapped up in getting it perfectly right. Successful projects make real-world pragmatic testing decisions, and so should you. If you see an opportunity to improve the tests and test suites, by all means, do it! Don't get hung up on naming and categorization, and refrain from passing judgment if the setup is not quite right; *code entropy* from Chapter 3 is a powerful force.

Test Tools

Test tools fall into several categories: test-writing tools, test execution frameworks, and code quality tools. Test-writing tools like mocking libraries help you write clean and efficient tests. Test frameworks help run tests by modeling a test's lifecycle from setup to teardown. Test frameworks also save test results, integrate with build systems, and provide other helpers. Code quality tools are used to analyze code coverage and code complexity, find bugs through static analysis, and check for style errors. Analysis tools are usually set up to run as part of a build or compile step.

Every tool added to your setup comes with baggage. Everyone must understand the tool, along with all of its idiosyncrasies. The tool might depend on many other libraries, which will further increase the complexity of the system. Some tools slow tests down. Therefore, avoid outside tools until you can justify the complexity trade-offs, and make sure your team is bought in.

Mocking

Mocking libraries are commonly used in unit tests, particularly in object-oriented code. Code often depends on external systems, libraries, or objects. *Mocks* replace external dependencies with stubs that mimic the interface provided by the real system. Mocks implement functionality required for the test by responding to inputs with hard-coded responses.

Eliminating external dependencies keeps unit tests fast and focused. Mocking remote systems allows tests to bypass network calls, simplifying the setup and avoiding slow operations. Mocking methods and objects allows developers to write focused unit tests that exercise just one specific behavior.

Mocks also keep application code from becoming riddled with test-specific methods, parameters, or variables. Test-specific changes are difficult to maintain, make code hard to read, and cause confusing bugs (don't add Boolean `isTest` method parameters!). Mocks help developers access protected methods and variables without modifying regular code.

While mocking is useful, don't overdo it. Mocks with complex internal logic make your tests brittle and hard to understand. Start with basic inline mocks inside a unit test, and don't write a shared mock class until you begin repeating mocking logic between tests.

An excessive reliance on mocks is a code smell that suggests tight code coupling. Whenever reaching for a mock, consider whether code could be refactored to remove the dependency on the mocked system. Separating computation and data transformation logic from I/O code helps simplify testing and makes the program less brittle.

Test Frameworks

Test frameworks help you write and execute tests. You'll find frameworks that help coordinate and execute unit tests, integration tests, performance tests, and even UI tests. Frameworks do the following:

- Managing test setup and teardown

- Managing test execution and orchestration
- Generating test result reports
- Providing tooling such as extra assertion methods
- Integrating with code coverage tools

Setup and teardown methods allow developers to specify steps, such as data structure setup or file cleanup, that need to be executed before or after each test or set of tests. Many test frameworks give multiple options for setup and teardown execution—before each test, before all tests in a file, or before all tests in a build. Read documentation before using setup and teardown methods to make sure you're using them correctly. Don't expect teardown methods to run in all circumstances. For example, teardown won't occur if a test fails catastrophically, causing the whole test process to exit.

Test frameworks help control the speed and isolation of tests through test orchestration. Tests can be executed serially or in parallel. Serial tests are run one after the next. Running one test at a time is safer because tests have less chance of impacting one another. Parallel execution is faster, but more error prone due to shared state, resources, or other contamination.

Frameworks can be configured to start a new process between each test. This further isolates tests, since each test will start fresh. Beware that starting new processes for each test is an expensive operation. See "Determinism in Tests" for more on test isolation.

Test reports help developers debug failed builds. Reports give a detailed readout of which tests passed, failed, or were skipped. When a test fails, reports show which assertion failed. Reports also organize logs and stack traces per test so developers can quickly debug failures. Beware, it's not always obvious where test results are stored—a summary is printed to the console, while the full report is written to disk. Look in test and build directories if you have trouble locating a report.

Code Quality Tools

Take advantage of tools that help you write quality code. Tools that enforce code quality rules are called *linters*. Linters run static analysis and perform style checks. Code quality monitoring tools report metrics such as complexity and test coverage.

Static code analyzers look for common mistakes like leaving file handles open or using unset variables. Static analyzers are particularly important for dynamic languages like Python and JavaScript, which do not have a compiler to catch syntax errors. Analyzers look for known code smells and highlight questionable code but are not immune to false positives, so you should think critically about problems reported by static analyzers and override false positives with code annotations that tell the analyzer to ignore particular violations.

Code style checkers ensure all source code is formatted the same way: max characters per line, proper indentation, camelCasing versus snake_casing, that sort of thing. A consistent style helps multiple programmers collaborate on a shared codebase. We highly recommend setting up your IDE so that all style rules are automatically applied.

Code complexity tools guard against overly complex logic by calculating *cyclomatic complexity*, or, roughly, the number of paths through your code. The higher your code's complexity, the more difficult it is to test, and the more defects it is likely to contain. Cyclomatic complexity generally increases with the size of the codebase, so a high overall score is not necessarily bad; however, a sudden jump in complexity can be cause for concern, as can individual methods of high complexity.

Code coverage tools measure how many lines of code were exercised by the test suite. If your change lowers code coverage, you should write more tests. Make sure that tests are exercising any new changes that you've made. Aim for reasonable coverage (the rule of thumb is between 65 and 85 percent). Remember that coverage alone isn't a good measure of test quality: it can be quite misleading, both when it is high and when it is low. Checking automatically generated code like scaffolding or serialization classes can create misleadingly low coverage metrics. Conversely, obsessively creating unit tests to get to 100 percent coverage doesn't guarantee that your code will integrate safely.

Engineers have a tendency to fixate on code quality metrics. Just because a tool finds a quality issue doesn't mean that it's actually a problem, nor does it mean that it's worth fixing immediately. Be pragmatic with codebases that fail quality checks. Don't let code get worse, but avoid disruptive stop-the-world cleanup projects. Use Chapter 4's section on technical debt as a guide to determine when to fix code quality issues.

Write Your Own Tests

You are responsible for making sure your team's code works as expected. Write your own tests; don't expect others to clean up after you. Many companies have formal *quality assurance (QA)* teams with varying responsibilities, including the following:

- Writing black-box or white-box tests
- Writing performance tests
- Performing integration, user acceptance, or system tests
- Providing and maintaining test tools
- Maintaining test environments and infrastructure
- Defining formal test certification and release processes

QA teams can help you verify your code is stable, but never "throw code over the fence" to have them do all of the testing. QA teams don't write unit tests anymore; those days are long gone. If you are in a company with a formal QA team, find out what they are responsible for and how to engage with them. If they're embedded within your team, they are likely attending scrum and sprint planning meetings (see Chapter 12 for more on Agile development). If they're a centralized organization, getting their help might require opening tickets or submitting some formal request.

Write Clean Tests

Write tests with the same care that you write other code. Tests introduce dependencies, require maintenance, and need to be refactored over time. Hacky tests have a high maintenance cost, which slows down future development. Hacky tests are also less stable and less likely to provide reliable results.

Use good programming practices on tests. Document how tests work, how they can be run, and why they were written. Avoid hard-coded values, and don't duplicate code. Use design best practices to maintain a separation of concerns and to keep tests cohesive and decoupled.

Focus on testing fundamental functionality, rather than implementation details. This helps when the codebase gets refactored, since tests will still run after the refactoring. If your test code is too tightly coupled with implementation particulars, changes to the main body of code will break tests. These breakages stop, meaning something broke, and just signal that the code changed. This does not provide value.

Keep test dependencies separate from your regular code dependencies. If a test requires a library to run, don't force the entire codebase to depend on the library. Most build and packaging systems will allow you to define dependencies specifically for tests; take advantage of this feature.

Don't Overdo Testing

Don't get swept up writing tests. It's easy to lose track of which tests are worth writing. Avoid chasing higher code coverage just to boost coverage metrics. Testing thin database wrappers, third-party libraries, or basic variable assignment is worthless even if it boosts coverage metrics. Focus on tests that have the largest effect on code risk.

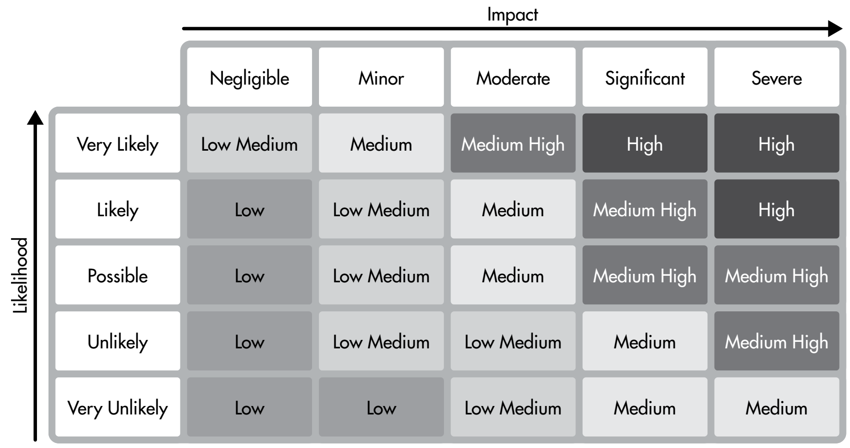
Use code coverage as a guide, not a rule. High code coverage does not guarantee correctness. Exercising code in a test counts toward coverage, but it doesn't mean that it was exercised usefully. It's entirely possible for critical errors to exist in codebases with 100 percent test coverage. Chasing a specific code coverage percentage is myopic.

Don't handcraft tests for autogenerated code such as web framework scaffolding or OpenAPI clients. If your coverage tools aren't configured to ignore generated code, the tools will report the code as untested. Fix the coverage tool configuration in such cases. Code generators are thoroughly tested, so testing generated code is a waste of time (unless you manually introduce changes to generated files, in which case you should test them). If for some reason you discover a real need to test generated code, figure out a way to add tests to the generator.

Focus effort on the highest value tests. Tests take time to write and maintain. Focusing on high-value tests yields the most benefit for the cost. Use a risk matrix to find areas to focus on. A *risk matrix* defines risk as the likelihood and impact of a failure.

Figure 6-1 is a sample risk matrix. The likelihood of a failure is measured on the y-axis, and the impact of the failure is measured on the x-axis. The intersection of the event's likelihood and impact defines its risk.

Tests shift code risk down the chart—more testing makes failures less likely. Focus on high-likelihood, high-impact areas of the code first. Low-risk or throwaway code like a proof of concept isn't worth testing.


Impact
Likelihood

	Negligible	Minor	Moderate	Significant	Severe
Very Likely	Low Medium	Medium	Medium High	High	High
Likely	Low	Low Medium	Medium	Medium High	High
Possible	Low	Low Medium	Medium	Medium High	Medium High
Unlikely	Low	Low Medium	Low Medium	Medium	Medium High
Very Unlikely	Low	Low	Low Medium	Medium	Medium

***Figure 6-1:** Risk matrix (Source: https://wiki.riskscape.org.nz/index.php/File:Matrix.png)*

Determinism in Tests

Deterministic code always produces the same output for the same input. By contrast, nondeterministic code can return different results for the same inputs. A unit test that invokes a call to a remote web service on a network socket is nondeterministic; if the network fails, the test will fail. Nondeterministic tests are a problem that plague many projects. It's important to understand why nondeterministic tests are bad, how to fix them, and how to avoid writing them.

Nondeterministic tests degrade test value. Intermittent test failures (known as *flapping tests*) are hard to reproduce and debug because they don't happen every run, or even every tenth run. You don't know whether the problem is with the test or with your code. Because flapping tests don't provide meaningful information, developers might ignore them and check in broken code as a result.

Intermittently failing tests should be disabled or fixed immediately. Fix a flapping test by running it repeatedly in a loop to reproduce the failure. IDEs have features to run tests iteratively, but a loop in a shell also works. Sometimes the nondeterminism is caused by interactions between tests or specific machine configurations—you'll have to experiment. Once you've reproduced the failure, you can fix it by eliminating the nondeterminism, or fixing the bug.

Nondeterminism is often introduced by improper handling of sleep, timeouts, and random number generation. Tests that leave side effects, or interact with remote systems, also cause nondeterminism. Escape nondeterminism by making time and randomness deterministic, cleaning up after tests, and avoiding network calls.

Seed Random Number Generators

Random number generators (RNGs) must be seeded with a value that dictates the random numbers you get from it. By default, random number generators will use the system clock as a seed. System clocks change over time, so two runs of a test with a random number generator will yield different results—nondeterminism.

Seed random number generators with a constant to force them to deterministically generate the same sequence every time it runs. Tests with constantly seeded generators will always pass or always fail.

Don't Call Remote Systems in Unit Tests

Remote system calls require network hops, which are unstable. Network calls can time out, which introduces nondeterminism into unit tests. A test might pass hundreds of times and then fail once due to network timeout. Remote systems are also unreliable; they can be shut off, restarted, or frozen. If a remote system is degraded, your test will fail.

Avoiding remote calls (which are slow) also keeps unit tests fast and portable. Speed and portability are critical for unit tests since developers run them frequently, and locally on development machines. Unit tests that depend on remote systems aren't portable because a host machine running a test must have access to the remote system, and remote test systems are often in internal integration test environments that aren't easily reachable.

You can eliminate remote system calls in unit tests by using mocks or by refactoring code so remote systems are only required for integration tests.

Inject Clocks

Code that depends on specific intervals of time can cause nondeterminism if not handled correctly. External factors like network latency and CPU speed affect how long operations take, and system clocks progress independently. Code that waits 500ms for something to happen is brittle. A test will pass if the code runs in 499ms but fail when it runs in 501ms. Static system clock methods like `now` or `sleep` signal that your code is time-dependent. Use injectable clocks rather than static time methods so you can control the timing that your code sees in a test.

The following `SimpleThrottler` Ruby class illustrates the problem. `SimpleThrottler` invokes a `throttle` method when the operation count exceeds a threshold, but the clock is not injectable.

```
class SimpleThrottler
  def initialize(max_per_sec=1000)
    @max_per_sec = max_per_sec
    @last_sec = Time.now.to_i
    @count_this_sec = 0
  end

  def do_work
    @count_this_sec += 1
    # ...
  end

  def maybe_throttle
    if Time.now.to_i == @last_sec and @count_this_sec > @max_per_sec
      throttle()
      @count_this_sec = 0
    end
    @last_sec = Time.now.to_i
  end

  def throttle
    # ...
  end
end
```

In the previous example, we can't guarantee that the `maybe_throttle` condition will be triggered in a test. Two consecutive operations can take an unbounded amount of time to run if the test machine is degraded, or the operating system decides to schedule the test process unfairly. Without control of the clock, it's impossible to test the throttling logic properly.

Instead, make system clocks injectable. Injectable clocks will let you use mocks to precisely control the passage of time in your tests.

```
class SimpleThrottler
  def initialize(max_per_sec=1000, clock=Time)
    @max_per_sec = max_per_sec
    @clock = clock
    @last_sec = clock.now.to_i
    @count_this_sec = 0
  end

  def do_work
    @count_this_sec += 1
    # ...
  end

  def maybe_throttle
    if @clock.now.to_i == @last_sec and @count_this_sec > @max_per_sec
      throttle()
      @count_this_sec = 0
    end
    @last_sec = @clock.now.to_i
  end

  def throttle
    # ...
  end
end
```

This approach, called *dependency injection*, allows tests to override clock behavior by injecting a mock into the clock parameter. The mock can return integers that trigger `maybe_throttle`. Regular code can default to the regular system clock.

Avoid Sleeps and Timeouts

Developers often use `sleep()` calls or timeouts when a test requires work in a separate thread, process, or machine to complete before the test can validate its results. The problem with this technique is that it assumes that the other thread of execution will finish in a specific amount of time, which is not something you can rely on. If the language virtual machine or interpreter garbage collects or the operating system decides to starve the process executing the test, your tests will (sometimes) fail.

Sleeping in tests, or setting long timeouts, also slows down your test execution and therefore your development and debugging process. If you have a test that sleeps for 30 minutes, the fastest your tests will ever execute is 30 minutes. If you have a high (or no) timeout, your tests can get stuck.

If you find yourself tempted to sleep or set a timeout in a test, see if you can restructure the test so that everything will execute deterministically. If not, that's okay, but make an honest effort. Determinism isn't always possible when testing concurrent or asynchronous code.

Close Network Sockets and File Handles

Many tests leak operating system resources because developers assume that tests are short-lived and that the operating system will clean everything when the test terminates. However, test execution frameworks often use the same process for multiple tests, which means leaked system resources like network sockets or file handles won't be immediately cleaned.

Leaked resources cause nondeterminism. Operating systems have a cap on the number of sockets and file handles and will begin rejecting new requests when too many resources are leaked. A test that is unable to open new socket or file handles will fail. Leaked network sockets also break tests that use the same port. Even if tests are run serially, the second will fail to bind to the port since it was opened but not closed previously.

Use standard resource management techniques for narrowly scoped resources, like try-with-resource, or with blocks. Resources that are shared among tests should be closed using setup and teardown methods.

Bind to Port Zero

Tests should not bind to a specific network port. Static port binding causes nondeterminism: a test that runs fine on one machine will fail on another if the port is already taken. Binding all tests to the same port is a common practice; these tests will run fine serially but fail when run in parallel. Test failures will be nondeterministic since the ordering of test execution isn't always the same.

Instead, bind network sockets to port zero, which makes the operating system automatically pick an open port. Tests can retrieve the port that was picked and use that value through the remainder of the test.

Generate Unique File and Database Paths

Tests should not write to statically defined locations. Data persistence has the same problem as network port binding. Constant file paths and database locations cause tests to interfere with each other.

Dynamically generate unique file names, directory paths, and database or table names. Dynamic IDs let tests run in parallel since they will all read and write to a separate location. Many languages provide utility libraries to generate temporary directories safely (like `tempfile` in Python). Appending UUIDs to file paths or database locations also works.

Isolate and Clean Up Leftover Test State

Tests that don't clean up state cause nondeterminism. State exists anywhere that data persists, usually in memory or on disk. Global variables like counters are common in-memory state, while databases and files are common disk state. A test that inserts a database record and asserts that one row exists will fail if another test has written to the same table. The same test will pass when run alone on a clean database. Leftover state also fills disk space, which destabilizes the test environment.

Integration test environments are complex to set up, so they are often shared. Many tests run in parallel, reading and writing to the same datastores. Be careful in such environments, as sharing resources leads to unexpected test behavior. Tests can affect each other's performance and stability. Shared datastores can cause tests to interfere with each other's data. Follow our guidance in the earlier "Generate Unique File and Database Paths" section to avoid collisions.

You must reset state whether your tests pass or not; don't let failed tests leave debris behind. Use setup and teardown methods to delete test files, clean databases, and reset in-memory test state between each execution. Rebuild environments between test suite runs to rid test machines of leftover state. Tools like containers or machine virtualization make it easy to throw away entire machines and start new ones; however, discarding and starting new virtual machines is slower than running setup and teardown methods, so such tools are best used on large groups of tests.

Don't Depend on Test Order

Tests should not depend on a specific order of execution. Ordering dependencies usually happen when a test writes data, and a subsequent test assumes the data is written. This pattern is bad for many reasons:

- If the first test breaks, the second will break, too.
- It's harder to parallelize the tests, since you can't run the second test until the first is done.
- Changes to the first test might accidentally break the second.
- Changes to the test runner might cause your tests to run in a different order.

Use setup and teardown methods to share logic between tests. Provision data for each test in the setup method, and clean up the data in the teardown. Resetting state between each run will keep tests from breaking each other when they mutate the state.

Do's and Don'ts

DO'S	DON'TS
DO use tests to reproduce bugs.	**DON'T** ignore the cost of adding new testing tools.
DO use mocking tools to help write unit tests.	**DON'T** depend on others to write tests for you.

DO'S	DON'TS
DO use code quality tools to verify coverage, formatting, and complexity.	**DON'T** write tests just to boost code coverage.
DO seed random number generators in tests.	**DON'T** depend solely on code coverage as a measure of quality.
DO close network sockets and file handles in tests.	**DON'T** use avoidable sleeps and timeouts in tests.
DO generate unique file paths and database IDs in tests.	**DON'T** call remote systems in unit tests.
DO clean up leftover test state between test executions.	**DON'T** depend on test execution order.

Level Up

Many (long) books have been written on software testing. We suggest targeting specific test techniques rather than reading exhaustive test textbooks.

Unit Testing by Vladimir Khorikov is the place to go if you want more on testing best practices. *Unit Testing* covers the philosophy of unit testing and common unit test patterns and anti-patterns. Despite its name, the book also touches on integration testing.

Kent Beck's *Test-Driven Development* covers TDD in detail. TDD is a great skill to have. If you find yourself in an organization that practices TDD, this book is a must.

Look at *The Pragmatic Programmer*'s section on property-based testing. We left property-based testing on the cutting room floor, but if you want to expand your capabilities, property-based testing is a great technique to learn.

Elisabeth Hendrickson's *Explore It!* discusses exploratory testing to learn about code. If you are dealing with complex code, *Explore It!* is a good read.