Code Complete, McConnell, Steve

# **Developer Testing**

TESTING IS THE MOST POPULAR quality-improvement activity—a practice supported by a wealth of industrial and academic research and by commercial experience.

* **Unit testing** is the execution of a complete class, routine, or small program that has been written by a single programmer or team of programmers, which is tested in isolation from the more complete system.
* **Component testing** is the execution of a class, package, small program, or other program element that involves the work of multiple programmers or programming teams, which is tested in isolation from the more complete system.
* **Integration testing** is the combined execution of two or more classes, packages, components, or subsystems that have been created by multiple programmers or programming teams. This kind of testing typically starts as soon as there are two classes to test and continues until the entire system is complete.
* **Regression testing** is the repetition of previously executed test cases for the purpose of finding defects in software that previously passed the same set of tests.
* **System testing** is the execution of the software in its final configuration, including integration with other software and hardware systems. It tests for security, performance, resource loss, timing problems, and other issues that can’t be tested at lower levels of integration.

Numerous additional kinds of testing are performed by specialized test personnel and are rarely performed by developers, including **beta tests, customer-acceptance tests, performance tests, configuration tests, platform tests, stress tests, usability tests**, and so on.

Testing is usually broken into two broad categories: **black-box testing and white-box (or glass-box) testing**.

* **“Black-box testing”** refers to tests in which the tester cannot see the inner workings of the item being tested.
* **“White-box testing”** refers to tests in which the tester is aware of the inner workings of the item being tested. This is the kind of testing that you as a developer use to test your own code.

**Testing is a means of detecting errors**. **Debugging** is a means of diagnosing and correcting the root causes of errors that have already been detected.

## Role of Developer Testing in Software Quality

Collaborative development practices in their various forms have been shown to find a higher percentage of errors than testing does.

* Testing is an important part of any software-quality program, and in many cases it’s the only part.
* A successful test is one that breaks the software. The goal of every other development activity is to prevent errors .
* Testing can never completely prove the absence of errors.
* Testing by itself does not improve software quality. If you want to improve your software, don’t just test more; develop better.
* Testing requires you to assume that you’ll find errors in your code. If you assume you won’t, you probably won’t. The main source of undetected errors was that erroneous output was not examined carefully enough. **The errors were visible, but the programmers didn’t notice them** (Myers 1978).

### How much time should be spent in developer testing on a typical project?

A commonly cited figure for all testing is 50 percent of the time spent on the project, but that’s misleading.

Testing alone takes less time. Second, that figure represents the amount of time that’s typically spent rather than the time that should be spent.

Depending on the project’s size and complexity, developer testing should probably take 8 to 25 percent of the total project time.

### What do you do with the results of developer testing?

Even if you never correct the defects that testing finds, testing describes how reliable the software is.

The results is that they can and usually do guide corrections to the software. Finally, over time, the **record of defects found through testing helps reveal the kinds of errors that are most common.**

## Recommended Approach to Developer Testing

A systematic approach to developer testing maximizes your ability to detect errors of all kinds with a minimum of effort.

* Test for each relevant requirement to make sure that the requirements have been implemented. **Plan the test cases for this step at the requirements stage or as early as possible—preferably before you begin writing the unit to be tested**.
* Test for each relevant design concern to make sure that the design has been implemented.
* Use “basis testing” . At a minimum, you should test every line of code.
* Use a **checklist of the kinds of errors you’ve made** on the project to date or have made on previous projects.
* Design the test cases along with the product. This can help avoid errors in requirements and design, which tend to be more expensive than coding errors.
* **Plan to test and find defects as early as possible because it’s cheaper to fix defects early.**

### Test First or Test Last?

Writing test cases first forces you to think at least a little bit about the requirements and design before writing code, which tends to produce better code.

The defect-cost increase graph—see Figure 3-1—suggests that writing test cases first will minimize the amount of time between when a defect is inserted into the code and when the defect is detected and removed.

* It’s hard to write a test case for a poor requirement.
* Test-first programming is one of the most beneficial software practices to emerge during the past decade and is a good general approach.

### Limitations with developer testing:

Developers tend to test for whether the code works (**clean tests**) rather than test for all the ways the code breaks (**dirty tests**). Immature testing organizations tend to have about five clean tests for every dirty test (creating 25 times as many dirty tests)

Developer testing tends to have an optimistic view of test coverage.

A better coverage standard is to meet what’s called “100% branch coverage,”

As valuable as developer testing is, it isn’t sufficient to provide adequate quality assurance on its own and should be supplemented with other practices, including independent testing and collaborative construction techniques.

Why isn’t it possible to prove that a program is correct by testing it? To use testing to prove that a program works, you’d have to test every conceivable input value to the program and every conceivable combination of input values. Even for simple programs, such an undertaking would become massively prohibitive.

Incomplete Testing Since exhaustive testing is impossible, practically speaking, the art of testing is that of picking the test cases most likely to find errors.

The easiest way to make sure that you’ve gotten all the bases covered is to calculate the number of paths through the program and then develop the minimum number of test cases that will exercise every path through the program.

Structured Basis Testing

If you use code coverage or logic coverage testing, you might create many more test cases than you would need to cover the same logic with structured basis testing.

Start with 1 for the straight path through the routine. Add 1 for each of the following keywords, or their equivalents: if, while, repeat, for, and, and or. Add 1 for each case in a case statement. If the case statement doesn’t have a default case, add 1 more.

Ease of testing is another good reason to keep your routines short and your boolean expressions simple.

Written with [StackEdit](https://stackedit.io/).