# Chapter 11

# Software Quality: Testing

> "... we have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing. We're more of a testing, a quality software organization than we're a software organization."
>
> — *Bill Gates, chairman and chief software architect of Microsoft, during an interview in 2002.*[1]

---

*Software testing* is the process of running a program in a controlled environment to check whether the program behaves as expected. The purpose of testing is to improve software quality. If a test fails—that is, the program does not behave as expected—there must be a fault, either in the program or in the specification of expected behavior. Either way, the test has provided feedback that can be used to remove the fault and improve quality.

At one time, coding and testing were distinct phases of software development. Testing accounted for half of the time and cost of development.[2] Coding was done by developers; testing was done by testers. Testers prided themselves on their ability to trigger failures and track down faults. Defects were more likely to be in the code for special or edge cases, since developers tended to pay more attention to the basic functionality.

Since then, test automation has blurred the distinction between coding and testing. Batteries of tests can be run automatically every time a change is made. Developers can thereby produce code of sufficient quality that the code

© 2018 Ravi Sethi. This working draft is intended for Computer Science students at the University of Arizona, Spring 2018.

```
1)  year = ORIGINYEAR; /* = 1980 */
2)  while (days > 365)
3)  {
4)      if (IsLeapYear(year))
5)      {
6)          if (days > 366)
7)          {
8)              days -= 366;
9)              year += 1;
10)         }
11)     }
12)     else
13)     {
14)         days -= 365;
15)         year += 1;
16)     }
17) }
```

Figure 11.1: Where is the fault?

can go straight from development to deployment. Automated tests are run as part of the deployment process to ensure that applications that used to work continue to work.

Although when and how tests are run may have changed, testing remains a significant part of software development. The principal techniques for defect detection and removal have remained the same. They are reviews, static analysis, and testing. Reviews and static analysis were covered in Chapter 10. This chapter explores the process of testing.

## 11.1 Overview of Testing

The code in Fig. 11.1 is from a digital music and video player. On December 31, 2008, owners of the player awoke to find that it froze on startup. On the last day of a leap year, the code in Fig. 11.1 loops forever.[3]

What went wrong?

**Example 11.1:** Suppose that the code in Fig. 11.1 is reached with variable `days` representing the current date as an integer. January 1, 1980 is represented by the integer 1; December 31, 1980 by 366 since 1980 was a leap year; and so on.

Variable `year` is initialized to 1980 on line 1. On exit from the while loop on lines 2-17, variable `year` represents the current year. The body of the loop computes the current year by repeatedly subtracting 366 for a leap year (line 8)

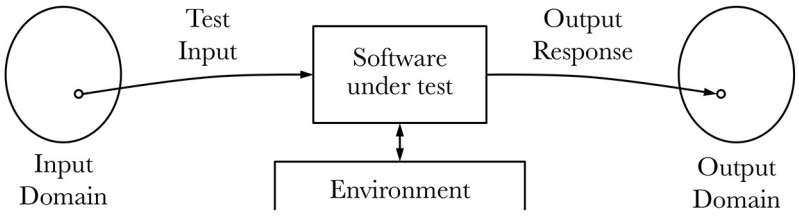


Figure 11.2: Software can be tested by applying an input stimulus and evaluating the output response.

or 365 for a non-leap year (line 14). Each subtraction is followed by a line that increments `year` (lines 9 and 15). In other words, the body of the loop counts down the days and counts up the years.

On the 366th day of 2008, a leap year, line 6 is eventually reached with value 366 for `days`. Control therefore loops back from line 6 to line 2 with `days` unchanged. Ad infinitum. □

How is it that "simple" bugs escape detection until there is an embarrassing product failure? The rest of this section explores the process of testing and its strengths and limitations.

### 11.1.1 Issues During Testing

The issues that arise during testing relate to the four main elements in Fig. 11.2:

- *Software Under Test.* The software under test can be a code fragment, a component, a subsystem, a self-contained program, or a complete hardware-software system.
- *Input Domain.* A tester selects an element of some input domain and uses it as test input.
- *Output Domain.* The output domain is the set of possible output responses or observable behaviors by the software under test. Examples of behaviors include producing integer outputs, as in Example 11.1, and displaying a web page.
- *Environment.* Typically, the software under test is not be self contained, so an environment is needed to provide the context for running the software.

If the software under test is a program fragment, the environment handles dependencies on the rest of the program. The environment also includes the operating system, libraries, and external services that may be running either locally or in the cloud. In the early stages of development, external services

Therac-25

can be simulated by dummy or mock modules with controllable behavior. For example, an external database can be simulated by a module that uses a local table seeded with known values.

**Example 11.2:** Suppose that the software under test is the code on lines 1-17 in Fig. 11.1. The input domain is the set of possible initial integer values for the variable `days`. The output domain is the set of possible final integer values for the variables `year` and `days`.

The code in Fig. 11.1 cannot be run as is, because it needs a definition for `ORIGNYEAR` and an implementation for function `IsLeapYear()`. These things must be provided by the environment. (We are treating the initial value of variable `days` as an input, so the environment does not need to provide a value for `days`.) □

The following questions capture the main issues that arise during testing:[4]

- How to stabilize the environment to make tests repeatable?
- How to select test inputs?
- How to evaluate the response to a test input?
- How to decide whether to continue testing?

### 11.1.2 Test Selection

The judicious selection of test inputs is a key problem during testing. Fortunately, reliable software can be developed without exhaustive testing on all possible inputs—exhaustive testing is infeasible.

#### The Input Domain

The term *test input* is interpreted broadly to include any form of input; e.g., a value such as an integer; a gesture; a combination of values and gestures; or an input sequence, such as a sequence of mouse clicks. In short, a test input can be any stimulus that produces a response from the software under test.

A set of tests is also known as a *test suite*.

The *input domain* is the set of all possible test inputs. For all practical purposes, the input domain is typically infinite. Variable `days` in Fig. 11.1 can be initialized to any integer value and, machine limitations aside, there is an infinite supply of integers.

Some faults are triggered by a mistimed sequence of input events. Therac-25 delivered a radiation overdose only when the technician entered patient-treatment data fast enough to trigger a fault; see Section 1.3. Other faults are triggered by an unfortunate combination of values. Avionics software is tested for interactions between multiple inputs; e.g., a decision may be based on data from a variety of sensors and a failure occurs only when the pressure crosses a threshold and the temperature is in a certain range.

Dijkstra, Edsger Wybe

It is important to test on both valid and invalid inputs, for the software must work as expected or valid inputs and do something sensible on invalid inputs. Crashing on invalid input is not sensible behavior.

Input domains can therefore consist a single values, (b) combinations of values, or (c) scenarios consisting of sequences of values.

#### Black-Box and White-Box Testing

During test selection, we can either treat the software under test as a black box or we can look inside the box at the source code. Testing that depends only on the software's interface is called *black-box* or *functional testing*. Testing that is based on knowledge of the source code is called *white-box* or *structural testing*. As we shall see in Section 11.2, white-box testing is used for smaller units of software and black-box testing is used for larger subsystems that are built up from the units.

Test design and selection is a theme that runs through this chapter.

### 11.1.3 Test Adequacy: Deciding When to Stop

Ideally, testing would continue until the desired level of software quality is reached. Unfortunately, there is no way of knowing when the desired level of quality is reached because, at any time during testing, there is no way of knowing how many more defects remain undetected. If a test fails—that is, the output does not match the expected response—the tester has discovered that there is a defect somewhere.

But, if the test passes, all the tester has learned is that the software works as expected on that particular test input. The software could potentially fail on some other input. As Edsger Dijkstra put it,

> "Testing shows the presence, not the absence of bugs."[5]

#### Stopping or Test-Adequacy Criteria

A *test adequacy* criterion is a measure of progress during testing. Adequacy criteria support statements of the form, "Testing is $x$% complete." Test adequacy criteria are typically based on three kinds of information: code coverage, input coverage, and defect-discovery data.

- *Code coverage* is the degree to which a construct in the source code is touched during testing. For example, statement coverage is the proportion of statements that are executed at least once during a set of tests. Code coverage is discussed in Section 11.3 on white-box testing.
- *Input coverage* is the degree to which a set of test inputs is representative of the whole input domain. For example, in Section 11.4 on black-box testing, the input domain will be partitioned into equivalence classes. Equivalence-class coverage is the proportion of equivalence classes that are represented in a test set.

Weyuker, Elaine Jessica

- *Defect-discovery data* includes data about the number and severity of the defects discovered in a given time interval. When combined with historical data from similar projects, the rate of defect discovery is sometimes used to make predictions about product quality.

Test adequacy criteria based on coverage and defect-discovery data are much better than arbitrary criteria such as stopping when time runs out or when a certain number of defects have been found. They cannot, however, guarantee the absence of bugs.

While testing alone is not enough, it can be a key component of an overall quality-improvement based on reviews, static analysis, and testing.[6]

### 11.1.4 Test Oracles: Evaluating the Response to a Test

Implicit in the above discussion is the assumption that we can readily tell whether an output is "correct;" that is, we can readily decide whether the output response to an input stimulus matches the expected output. This assumption is called the oracle assumption.

The *oracle assumption* has two parts:

1. There is a specification that defines the correct response to a test input.

2. There is a mechanism to decide whether or not a response is correct. Such a mechanism is called an *oracle*.

Most of the time, there is an oracle, human or automated. For values such as integers and characters, all an oracle may need to do is to compare the output with the expected value. An oracles based on a known comparison can be easily automated.

Graphical and audio/video interfaces may require a human oracle. For example, how do you evaluate a text-to-speech system? It may require a human to decide whether the spoken output sounds natural to a native speaker.

#### Questioning the Oracle Assumption

The oracle assumption does not always hold. A test oracle may not be readily available, or may be nontrivial to construct.

**Example 11.3:** Elaine Weyuker gives the example of a major oil company's accounting software, which "had been running without apparent error for years."[7] One month, it reported $300 for the company's assets, an obviously incorrect output response. This is an example of knowing that a response is incorrect, without knowing the right response.

There was no test oracle for the accounting software. Even an expert could not tell whether "$1,134,906.43 is correct and $1,135,627.85 is incorrect." □

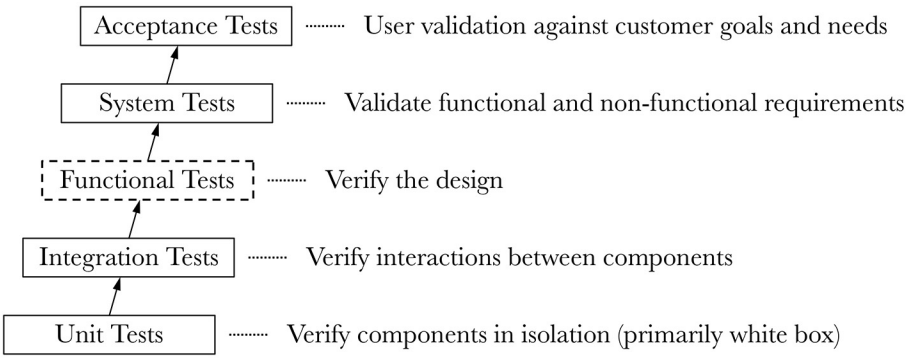


Figure 11.3: Levels of testing. Functional tests may be merged into system tests; hence the dashed box.

**Example 11.4:** Consider a program that takes as input an integer $n$ and produces as output the $n$th prime number. On input 1 the program produces 2, the first prime number; on 2 it produces 3; on 3 it produces 5; and so on. On input 1000, it produces 7919 as output.

Is 7919 the 1000th prime? (Yes, it is.)

It is nontrivial to create an oracle to decide if a number $p$ is the $n$th prime number.[8] □

## 11.2 Levels of Testing

Testing becomes more manageable if the problem is partitioned: bugs are easier to find and fix if components are debugged before they are assembled into a larger system. A components-before-systems approach motivates the levels of testing in Fig. 11.3. Testing proceeds from bottom to top; the size of the software under test increases from bottom to top.

Each level of testing plays a different role. The terms "verify" and "validate" were introduced as follows in Section 10.2:

*Validation*: "Am I building the right product?"
*Verification*: "Am I building the product right?"

The top two levels in Fig. 11.3 validate that customer needs and requirements are met. The lower levels verify the implementation. System and functional testing may be combined into a single level that tests the behavior of the system; hence the dashed box for functional tests. The number of levels varies from project to project, depending on the complexity of the software and the importance of the application.[9]

Based on data from hundreds of companies, each level catches about one in three defects.[10]

Beck, Kent
Gamma, Erich

### 11.2.1 Unit Testing

A *unit* of software is a logically separate component that can be tested by itself. It may be a module or part of a module. *Unit testing* verifies a unit in isolation from the rest of the system. With respect to the overview of testing in Fig. 11.2, the environment simulates just enough of the rest of the system to allow the unit to be run and tested.

Unit testing is primarily white box testing, where test selection is informed by the source code of the unit. White-box testing is discussed in Section 11.3.

#### xUnit: Automated Unit Testing

Convenient automated unit testing profoundly changes software development. A full suite of tests can be run automatically at any time to verify the code. Changes can be made with reasonable assurance that the changes will not break any existing functionality. Code and tests can be developed together; new tests can be added as development proceeds. In fact, automated tests enable test-first or test-driven development, where tests are written first and then code is written to pass the tests.

Convenience was the number one goal for *JUnit*, a framework for automated testing of Java programs. Kent Beck and Erich Gamma wanted to make it so convenient that "we have some glimmer of hope that developers will actually write tests."[11]

JUnit quickly spread. It inspired unit testing frameworks for other languages, including CUnit for C, CPPUnit for C++, PyUnit for Python, JSUnit for JavaScript, and so on. This family of testing frameworks is called *xUnit*.

An xUnit test proceeds as follows:

```
set up the environment;
run test;
tear down the environment;
```

From Section 11.1, the environment includes the context that is needed to run the software under test. For a Java program, the context includes values of variables and simulations of any constructs that the software relies on.

**Example 11.5:** The pseudo-code in Fig. 11.4(a) shows a class `Date` with a method `getYear()`. The body of `getYear()` is not shown—think of it as implementing the year calculation in Fig. 11.1.

The code in Fig. 11.4(b) sets up a single JUnit test for `getYear()`. The annotation `@Test` marks the beginning of a test. The name of the test is `test365`. A descriptive name is recommended, for readability of messages about failed tests. Simple tests are recommended to make it easier to identify faults.

The test creates object `date` and calls `getYear(365)`, where 365 represents December 31, 1980. JUnit supports a range of assert methods; `assertEquals()` is an example. If the computed value `year` does not equal 1980, the test fails, and JUnit will issue a descriptive message.

```
public class Date {
   ...
   public int getYear(...) {
      ...
   }
   ...
}
```

(a) Software Under Test

```
public class DateTest {
   @Test
   public void test365() {
      Date date = new Date();
      int year = date.getYear(365);
      assertEquals(year, 1980);
    }
}
```

(a) A JUnit 4 Test

Figure 11.4: A JUnit test.

For more information about JUnit, visit `junit.org`. □

### 11.2.2 Integration Testing

*Integration testing* verifies interactions between the components of a subsystem. Integration testing is typically black-box testing.

Prior unit testing increases the likelihood that a failure during integration testing is due to interactions between components instead of being due to a fault within some component.

With *big bang* integration testing, the whole system is assembled all at once from individually unit tested components. *Incremental* integration testing is a better approach: interactions are verified as components are added, one or more at a time, to a previously tested set of components.

#### Dependencies Between Modules

Incremental integration testing must deal with dependencies between modules.

**Example 11.6:** In a model-view-controller architecture, the view displays information that it gets from the model. The view depends on the model, but not the other way around.

The model in Example 8.5 held information about a picture of the Mona Lisa, including a digital photo and the height, width, and resolution of the photo. There were two views: one displayed the digital photo; the other displayed the height, width, and resolution of the photo.

Both views got their information from the model, so they depended on the model. The model, however, was not dependent on the views. □

Dependencies between modules can be defined in terms of a uses relationship: module $M$ *uses* module $N$, if $N$ must be present and satisfy its specification for $M$ to satisfy its specification.[12] Note that the used module need not be a subcomponent of the using module. In Example 11.6, the views used the model, but the model was not a subcomponent of either view.