If you took CS 125, you've probably run test cases written by other people. However, in this class, and in the real world, no one is going to write your test cases for you. In this reading, we'll discuss why tests are important, and how to write good tests.

**Validation**

Testing is an example of a more general process called validation. The purpose of validation is to uncover problems in a program and thereby increase your confidence in the program's correctness.

**Why software testing is hard**

Here are some approaches that unfortunately don't work well in the world of software.

Exhaustive testing is infeasible. The space of possible test cases is generally too big to cover exhaustively. Imagine exhaustively testing a 32-bit floating-point multiply operation, a*b. There are $2^{64}$ test cases!

Haphazard testing ("just try it and see if it works") is less likely to find bugs, unless the program is so buggy that an arbitrarily-chosen input is more likely to fail than to succeed. It also doesn't increase our confidence in program correctness.

Random or statistical testing doesn't work well for software. Other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer the defect rate for the whole production lot. Physical systems can use many tricks to speed up time, like opening a refrigerator 1000 times in 24 hours instead of 10 years. These tricks give known failure rates (e.g. mean lifetime of a hard drive), but they assume continuity or uniformity across the space of defects. This is true for physical artifacts.

But it's not true for software. Software behavior varies discontinuously and discretely across the space of possible inputs. The system may seem to work fine across a broad range of inputs, and then abruptly fail at a single boundary point. The famous Pentium division bug affected approximately 1 in 9 billion divisions. Stack overflows, out of memory errors, and numeric overflow bugs tend to happen abruptly, and always in the same way, not with probabilistic variation. That's different from physical systems, where there is often visible evidence that the system is approaching a failure point (cracks in a bridge) or failures are distributed probabilistically near the failure point (so that statistical testing will observe some failures even before the point is reached).

Instead, test cases must be chosen carefully and systematically. Techniques for systematic testing are the primary focus of this reading.

**Test-first programming**

Before we dive in, we need to define some terms:

- A module is a part of a software system that can be designed, implemented, tested, and reasoned about separately from the rest of the system. In this reading, we'll focus on modules that are functions, represented by Java methods. In future readings we'll broaden our view to think about larger modules, like a class with multiple interacting methods.
- A specification (or spec) describes the behavior of a module. For a function, the specification gives the types of the parameters and any additional constraints on them (e.g. sqrt's parameter must be nonnegative). It also gives the type of the return value and how the return value relates to the inputs. In Java code, the specification consists of the method signature and the comment above it that describes what it does.
- A module has an implementation that provides its behavior, and clients that use the module. For a function, the implementation is the body of the method, and the clients are other code that calls the method. The specification of the module constrains both the client and the implementation. We'll have much more to say about specifications, implementations, and clients a few classes from now.
- A test case is a particular choice of inputs, along with the expected output required by the specification.
- A test suite is a set of test cases for a module.

In test-first-programming, you write the spec and the tests before you even write any code. The development of a single function proceeds in this order:

1. Spec: Write a specification for the function.
2. Test: Write tests that exercise the specification.
3. Implement: Write the implementation.

Once your implementation passes the tests you wrote, you're done.

It turns out that this is a good pattern to follow when designing a program from scratch. The biggest benefit of test-first programming is safety from bugs. Don't leave testing until the end of development, when you have a big pile of unvalidated code. Leaving testing until the end only makes debugging longer and more painful, because bugs may be anywhere in your code. It's far more pleasant to test your code as you develop it.

(For this week's assignment, TicTacToe, we're giving you the spec of the function that you have to implement, so you can start at step 2. But in the future, assignments will be more complex and open-ended, so it will be up to you to determine how to decompose all of the functionality of your program into modules, and write specs for each of those modules.)

**Systematic testing**
Rather than exhaustive, haphazard, or randomized testing, we want to test systematically. Systematic testing means that we are choosing test cases in a principled way, with the goal of designing a test suite with three desirable properties:

- Correct. A correct test suite is a legal client of the specification, and it accepts all legal implementations of the spec without complaint. This gives us the freedom to change how the module is implemented internally without necessarily having to change the test suite.
- Thorough. A thorough test suite finds actual bugs in the implementation, caused by mistakes that programmers are likely to make.
- Small. A small test suite, with few test cases, is faster to write in the first place, and easier to update if the specification evolves. Small test suites are also faster to run. You will be able to run your tests more frequently if your test suites are small and fast.

By these criteria, exhaustive testing is thorough but infeasibly large. Haphazard testing tends to be small but not thorough. Randomized testing can achieve thoroughness only at the cost of large size.

Designing a test suite for both thoroughness and small size requires having the right attitude. Normally when you're coding, your goal is to make the program work. But as a test suite designer, you want to make it fail. That's a subtle but important difference. A good tester intentionally pokes at all the places the program might be vulnerable, so that those vulnerabilities can be eliminated.

The need to adopt a testing attitude is another argument for test-first programming. It is all too tempting to treat code you've already written as a precious thing, a fragile eggshell, and test it very lightly just to see it work. For thorough testing, though, you have to be brutal. Test-first programming allows you to put on your testing hat, and adopt that brutal perspective, before you've even written any code.

**Choosing test cases by partitioning**

Creating a good test suite is a challenging and interesting design problem. We want to pick a set of test cases that is small enough to be easy to write and maintain and quick to run, yet thorough enough to find bugs in the program.

To do this, we divide the input space into subdomains, each consisting of a set of inputs. (The name subdomain comes from the fact that it is a subset of the domain, another name for the input space of a mathematical function.) Taken together, the subdomains form a partition: a collection of disjoint sets that completely covers the input space, so that every input lies in exactly one subdomain. Then we choose one test case from each subdomain, and that's our test suite.

The idea behind subdomains is to divide the input space into sets of similar inputs on which the program has similar behavior. Then we use one representative of each set. This approach makes the best use of limited testing resources by choosing dissimilar test cases, and forcing the testing to explore areas of the input space that random testing might not reach.

**Example: max()**

```
/**
 * ...
 * @param a  an argument
 * @param b  another argument
 * @return the larger of a and b.
 */
public static int max(int a, int b)
```

Here, we have a two-dimensional input space, consisting of all the pairs of integers (a,b). Now let's partition it. From the specification, it makes sense to choose these subdomains:

- a < b
- a > b

because the spec calls for different behavior on each one. But we can't stop there, because these subdomains are not yet a partition of the input space. A partition must completely cover the set of possible inputs. So we need to add:

- a = b

Our test suite might then be:

- (a, b) = (1, 2) to cover a < b
- (a, b) = (10, -8) to cover a > b
- (a, b) = (9, 9) to cover a = b

**Include boundaries in the partition**

Bugs often occur at boundaries between subdomains. Some examples:

- 0 is a boundary between positive numbers and negative numbers
- the maximum and minimum values of numeric types, like int or double
- emptiness for collection types, like the empty string, empty list, or empty set
- the first and last element of a sequence, like a string or list

Why do bugs often happen at boundaries? One reason is that programmers often make off-by-one mistakes, like writing <= instead of <, or initializing a counter to 0 instead of 1. Another is that some boundaries may need to be handled as special cases in the code. Another is that boundaries may be places of discontinuity in the code's behavior. When an int variable grows beyond its maximum positive value, for example, it abruptly becomes a negative number.

**Automated unit testing with JUnit**

A well-tested program will have tests for every individual module that it contains. A test that tests an individual module, in isolation if possible, is called a unit test.

JUnit is a widely-adopted Java unit testing library, and we will use it for all Java assignments in CS 126. A JUnit unit test is written as a method preceded by the annotation @Test. A unit test method typically contains one or more calls to the module being tested, and then checks the results using assertion methods like assertEquals, assertTrue, and assertFalse.

For example, the tests we chose for max above might look like this when implemented for JUnit:

```
public class MaxTest {
  @Test
  public void testALessThanB() {
      assertEquals(2, Math.max(1, 2));
  }

  @Test
  public void testBothEqual() {
      assertEquals(9, Math.max(9, 9));
  }

  @Test
  public void testAGreaterThanB() {
      assertEquals(10, Math.max(10, -9));
  }
}
```

Note that the order of the parameters to assertEquals is important. The first parameter should be the expected result, usually a constant, that the test wants to see. The second parameter is the actual result, what the code actually does. If you switch them around, then JUnit will produce a confusing error message when the test fails. All the assertions supported by JUnit follow this order consistently: expected first, actual second. An assertion can also take an optional message string as the last argument, which you can use to make the test failure clearer.

If an assertion in a test method fails, then that test method returns immediately, and JUnit records a failure for that test. A test class can contain any number of @Test methods, which are run independently when you run the test class with JUnit. Even if one test method fails, the others will still be run.

**Documenting your testing strategy**

It's a good idea to write down (in comments) the testing strategy you used to create a test suite: the partition, and which subdomain of the partition each test case was chosen to cover. Writing down the strategy makes the thoroughness of your test suite much more visible to the reader.

You should do this in your TicTacToe assignment! Not only will it help others understand your testing strategy; it will also force you to make sure that you've covered all of the different types of inputs/outcomes.

Here is an example of how to do this:

```
public class MaxTest {
  /*
   * Testing strategy
   *
   * Partitions max(a,b) as follows:
   *    a < b, a > b, a = b
   */

  // covers a < b
  @Test
  public void testALessThanB() {
      assertEquals(2, Math.max(1, 2));
  }

  // repeat for other test cases
}
```

**Black box and glass box testing**

Recall from above that the specification is the description of the function's behavior — the types of parameters, type of return value, and constraints and relationships between them.

Black box testing means choosing test cases only from the specification, not the implementation of the function. That's what we've been doing in our examples so far. We partitioned and looked for boundaries in abs, max, and multiply without looking at the actual code for these functions. In fact, following the test-first programming approach, we hadn't even written the code for these functions yet.

Glass box testing means choosing test cases with knowledge of how the function is actually implemented. For example, if the implementation selects different algorithms depending on the input, then you should partition around the points where different algorithms are chosen. If the implementation keeps an internal cache that remembers the answers to previous inputs, then you should test repeated inputs.

When doing glass box testing, you must take care that your test cases don't require specific implementation behavior that isn't specifically called for by the spec. (Keep this in mind when you're answering the questions on PrairieLearn.) For example, if the spec says "throws an exception if the input is poorly formatted," then your test shouldn't check specifically for a NullPointerException just because that's what the current implementation does. The specification in this case allows any exception to be thrown, so your test case should likewise be general in order to be correct, and preserve the implementor's freedom. We'll have much more to say about this in the class on specs.

**Automated testing and regression testing**
Nothing makes tests easier to run, and more likely to be run, than complete automation. Automated testing means running the tests and checking their results automatically.

The code that runs tests on a module is a test driver (also known as a test harness or test runner). A test driver should not be an interactive program that prompts you for inputs and prints out results for you to manually check. Instead, a test driver should invoke the module itself on fixed test cases and automatically check that the results are correct. The result of the test driver should be either “all tests OK” or “these tests failed: …” A good testing framework, like JUnit, allows you to build and run this kind of test driver, with a suite of automated tests.

Note that automated testing frameworks like JUnit make it easy to run the tests, but you still have to come up with good test cases yourself. Automatic test generation is a hard problem, still a subject of active computer science research.

Once you have test automation, it’s very important to rerun your tests when you modify your code. Software engineers know from painful experience that any change to a large or complex program is dangerous. Whether you’re fixing another bug, adding a new feature, or optimizing the code to make it faster, an automated test suite that preserves a baseline of correct behavior – even if it’s only a few tests – will save your bacon. Running the tests frequently while you’re changing the code prevents your program from regressing — introducing other bugs when you fix new bugs or add new features. Running all your tests after every change is called regression testing.

Whenever you find and fix a bug, take the input that elicited the bug and add it to your automated test suite as a test case. This kind of test case is called a regression test. This helps to populate your test suite with good test cases. Remember that a test is good if it elicits a bug — and every regression test did in one version of your code! Saving regression tests also protects against reversions that reintroduce the bug. The bug may be an easy error to make, since it happened once already.

This idea also leads to test-first debugging. When a bug arises, immediately write a test case for it that elicits it, and immediately add it to your test suite. Once you find and fix the bug, all your test cases will be passing, and you’ll be done with debugging and have a regression test for that bug.

In practice, these two ideas, automated testing and regression testing, are almost always used in combination. Regression testing is only practical if the tests can be run often, automatically. Conversely, if you already have automated testing in place for your project, then you might as well use it to prevent regressions. So automated regression testing is a best-practice of modern software engineering.

**How testing relates to the themes of CS 126:**

- Safe from bugs. Testing is about finding bugs in your code, and test-first programming is about finding them as early as possible, right after you introduce them.
- Easy to understand. Systematic testing with a documented testing strategy makes it easier to understand how test cases were chosen and how thorough a test suite is.
- Ready for change. Correct test suites only depend on behavior in the spec, which allows the implementation to change within the confines of the spec. We also talked about automated regression testing, which helps keep bugs from coming back when changes are made to code.