The purpose of this course is to discuss a variety of techniques and methods that can be used to find faults in, or to enhance our confidence in the correctness of, a given computer program.

It is assumed that there exists a specification of the program in question. According to the IEEE standard, an execution result that violates the program specification is said to be a failure. A portion of the program that leads to a failure is called a fault. A fault may produce more than one kind of failure. A failure, however, is uniquely associated with a fault.

To find faults in a program, we must be able to correctly identify a program failure. That is not always easy to do because the program specification might be ambiguous, inconsistent, or difficult to interpret. Symbolic logic, consisting of the propositional calculus (Lectures 1 and 2) and first-order predicate calculus (Lectures 3 and 4), may be used to alleviate the problem.

The concepts of recursive definition and proof by mathematical induction (Lecture 5) are also important in determining the correctness of a program because we often have to define a set of (potentially) infinite number of objects with a description of finite length, and to argue that our guesses based on a finite number of observations are correct.

With the foregoing preparation in logic, the students by now should be able to follow discussions on how to prove the correctness of a program. I will show the students how to construct a correctness proof in a bottom-up (Lecture 6) and top-down (Lecture 7) manner. The purpose here is to point out the technical bottle necks that make the technique impractical, to introduce the concept of loop invariant that can be used to communicate the intended function of a loop construct, and to demonstrate how the concept of weakest precondition can be used in the process of test-case generation (Lecture 8).

When we attempt to analyze a program for its functionality or correctness, we often have to do it pathwise, i.e., to work with a subprogram consisting of one or more execution path. For this purpose, I will present a graph analysis method in which a set of paths is represented as a regular expression (Lecture 9). Furthermore, I will introduce the concepts of state constraint and pathwise decomposition (Lectures 10 and 11) so that we can systematically decompose a program into a set of subprograms, rigorously analyze each of which to determine the condition under which it will be executed, and the function it will perform in the process.

Now we are in a position to discuss different methods for testing and analyzing programs. For each method discussed, the students are expected to know in depth what is the basic idea involved, what is its capability and limitation, what it takes to use it in practice, and if it is not practical, why that is so.

Generally speaking, we do testing for two different purposes. One is to discover latent error (debug testing), and the other is to assess the reliability of the program (operational testing).

We shall discuss debug testing first. In debug testing, the main problem is find test cases (i.e., the input data used to test-execute the program) that have a high probability of revealing an error.
Observe that not every component of a program is involved in a particular execution. Now if a component contains an error and is not involved in a test execution, that error will never be revealed by that test. Thus, a way to increase the test effectiveness is to select test cases to "exercise" as many components as possible. Different definitions of component lead to the development of different test-case selection methods. A component can be defined based on the source code or specification.

Code-based test-case selection methods include:
- statement testing (Lecture 12)
- branch testing (Lecture 12)
- path testing (Lecture 13)
- data-flow testing (Lecture 14)
- domain strategy testing (Lecture 15)
- program mutation (Lecture 16)

Specification-based test-case selection methods include:
- error guessing (Lecture 17)
- equivalence partitioning (Lecture 17)
- boundary-value analysis (Lecture 17)
- cause-effect graphing (Lecture 18)

If the purpose of testing is to assess the reliability of the program, then we do operational testing (Lecture 19) by selecting test cases based on the operational profile. Basically, an operational profile is a list of program inputs and their probabilities of being used in production runs. If we use a set of inputs with an aggregate probability of x percent, and if the resulting outputs are all correct, then we could reasonably assert that the program will work correctly for x percent of the time in production runs. That is the gist of operational testing. To do operational testing in practice, the main job is to construct a reasonably accurate operational profile. We use an example to demonstrate how it can be done.

Operational testing can be done for the purpose of reliability enhancement as well. This possibility is of interest to the practitioners because it cost much less to find test cases in operational testing. Assume that a fault detected in operational testing is fixed immediately, and the fix introduces no new fault into the program. It is easy to see that, increasingly, the cost of debugging becomes higher, and the benefit of improvement in reliability lesser. At some point, the net benefit of this effort will drop below an acceptable value. Obviously that is a good time to stop testing. The question is, can we estimate the cost and benefit of software testing? We will study a theoretical model developed for this purpose (Lecture 20).

By this time, the students may start to realize that what we can do with testing, regardless of the methods used, is rather limited. Before we proceed to explore the question of what else we can do, we would like to examine two theoretical treatments that address the following questions.

First, can we construct a test for any program so that its success constitutes a direct proof of the correctness? The answer is negative (Lecture 21), but we were able to characterize such a test.
Second, is an operational test more cost effective than a debug test? It turns out that there is no clear answer to this question (Lecture 22).

Is there anything else that we can do, in addition to testing, to detect latent faults in a program, or to increase our confidence in a program? The answer is affirmative. First, we can perform static analysis. Second, we can use program instrumentation to facilitate test process and increase the error-detection capability of a test.

By static analysis we mean to examine the source code systematically without executing it. We shall discuss a method for detecting data-flow anomaly (Lecture 23), which is a symptom of possible programming error. We shall discuss the technique of symbolic execution that can be used to verify the computation performed by a straight-line code (Lecture 24). We shall also discuss the concept of program slicing that can be used to facilitate program understanding (Lecture 25).

By program instrumentation we mean to insert additional code into the program to be tested for the purpose of facilitating the test process or generating additional information for error detection. We shall discuss how to instrument a program (Lectures 26-27) to monitor test coverage automatically, to measure the effectiveness of test cases used, to detect data-flow anomalies, to reveal errors through assertion checking, and to generate symbolic traces (i.e., to perform pathwise decomposition) automatically. The symbolic trace of an execution path can be analyzed to provide vital information about a test, such as the condition under which the execution path will be traversed, and the computation to be performed during the execution.

We shall discuss test strategies that have been proved to be effective at different stages in the software life cycle (Lecture 28), such as the strategy to be used whenever a program has been modified to remove a fault, or when components are being integrated to form the complete system.

Finally, we note that many of the methods and techniques discussed above have been developed for traditional programs. In recent years, however, we use object-oriented technology to develop most new programs. We shall discuss how an object-oriented program differs from a traditional program from a tester's point of view, which part of the existing testing techniques remain applicable, and where new methods are needed to deal with testing problems that are unique to object-oriented programs (Lectures 29-30).