As with previous chapters, we are going to lay out the learning objectives for this chapter, and split them into what you should get on the first reading vs the second reading.

1 First Reading: Understand Key Ideas

On your first reading, the goal is to get the big ideas, and build the mental scaffolding in which to hold the details on your second reading—the learning objectives for your first reading are generally **Remember** and **Understand** objectives. If you are missing a few details (but not completely lost), it is ok to move ahead, and come back to them the next time through—you might ask questions on the course forums too! Note you should **sleep at least one night between the first and second reading** to give your brain time to process the information.

- **Fixing Your Code: Intro**
  - **Define** *testing*.
  - **Define** *debugging*.
  - **Explain** the difference and relationship between testing and debugging.
  - **Explain** how investing time on testing can save you time overall.
    - **Note**: if you really internalize this learning objective and put it to use by good incremental development, you will save yourself a LOT of time in many courses (and be an overall better software engineer).
- **Step 6: Test Your Code**
  - **Explain** what makes a good test case and why
  - **State** possible catastrophic consequences of buggy software.
  - **Define** *incremental testing*.
- **Black Box Testing**
  - **Define** *black box testing*.
  - **Define** *edge case*.
  - **Define** *corner case*.
  - **Define** *equivalence class*.
  - **Explain** why equivalence classes are useful in testing.
  - **Define** *reproduce* (a bug).
  - **Explain** why your tests should cover every error case.
  - **Explain** why and how your error tests should cover "too many" and "too few" whenever applicable.
  - **Explain** why any one test case can only test one error condition.
  - **Explain** why you should test at the boundary of validity.
  - **Explain** why you should think about unusual/special cases.

- **Explain** why you should think about how a programmer might have misunderstood the requirements.
- **Explain** why you should consider types, and what kinds of problems the wrong type might cause.
- **Define** *off-by-one error*.
- **Explain** why you should think about off-by-one errors.
- **Explain** how to think about inputs to test for a particular mistake you have in mind.
- **Explain** the idea of covering common categories based on the input type.
- **Explain** how to approach making a table of input/output categories and why it can be useful.

- **White Box Testing**
  - **Define** *white box testing*.
  - **Define** *test coverage*.
  - **List** the three common levels of code coverage.
  - **Define** *statement coverage*.
  - **Explain** why statement coverage is a bare minimum of test coverage.
  - **Define** *decision coverage* (also called "branch coverage").
  - **Define** *control flow graph*
  - **Explain** how a set of test cases can give statement coverage without giving decision coverage.
  - **Define** *path coverage*.
  - **Explain** why path coverage is often difficult to obtain in real code.
  - **Explain** how to determine what level of coverage you need.
  - **State** how many test cases you need to ensure code is correct.

- **Generating Tests**
  - **Explain** why you might want to algorithmically generate tests.
  - **Define** *random testing*.
  - **Explain** a major difficulty in algorithmically generating tests.
  - **Explain** some ways to overcome this difficulty.
  - **Define** *test harness*.

- **Asserts**
  - **Define** *invariant*.
  - **Define** *assert statement*.
  - **Define** *fail fast*.
  - **Explain** why assert statements are useful.

- **Explain** why giving the wrong answer is typically worse than crashing.
- **Explain** why you should not generally worry about asserts slowing your program down.

- **Regression Testing and Code Review**
  - Note: we briefly mention the topics in 6.1.5 and 6.1.6 as they are important. However, in-depth coverage (and higher-level learning objectives) are left to a later Software Engineering class.
  - **Define** *regression testing*.
  - **Define** *code review*

- **Testing and Types**
  - **Explain** the difference between testing individual functions and testing the entire program.
  - **Explain** what happens when you are testing an individual function (by calling it directly) and you pass the wrong type of data.
  - **Explain** what happens when you are testing an individual function (by calling it directly) and you pass constants that are out of the range of the parameter type.
  - **Explain** how command line arguments are passed to programs.
  - **Explain** the difference between the textual representation of a number and the number itself.

- **Know The Right Behavior**
  - **Recognize** that in the real world requirements come from customers, and customers are not always great at explaining their requirements.
  - **Explain** tradeoffs in being more permissive versus more restrictive in what inputs are legal.

- **Step 7: Debug Your Code**
  - **Define** *ad hoc*.
  - **Explain** what *ad hoc* debugging is, and why it is a poor approach.
  - **Know** that debugging should be done as an application of the scientific method.
  - **Explain** the first step of the scientific method (Observe a Phenomenon) and how it relates to debugging.
  - **Explain** the second step of the scientific method (Ask a Question) and how it relates to debugging.
  - **Explain** the third step of the scientific method (Gather Information and Apply Expert Knowledge) and how it relates to debugging.
  - **Explain** what kind of expert knowledge you use in debugging a program.
  - **Explain** how to use print statements to gather information.
  - **Define** *debugger*.

- **Explain** how to use gdb to gather information.
  - Note: we recommend reading D.2 for the details of gdb after you finish this chapter. We strongly recommend practicing with gdb and getting proficient at it. It takes some work to learn it, but being good at gdb saves you a lot of time in the long run.
- Note: we assume you already know what a *hypothesis* is, but if not you should **define** it before proceeding.
- **Explain** the fourth step of the scientific method (Form a Hypothesis) and how it relates to debugging.
- **Explain** what it means for a hypothesis to be *testable*.
- **Explain** what it means for a hypothesis to be *actionable*.
- **Explain** the fifth step of the scientific method (Test the Hypothesis) and how it relates to debugging.
- Note: we assume you already know what a *false economy* is, but if not you should **define** it before proceeding.
- **Explain** why skipping over testing a hypothesis is a false economy.
- **List** four ways to test a hypothesis about your code.
- **Explain** each of these four ways to test a hypothesis about your code.
- **Explain** The Poker Player's Fallacy, and how it relates to making changes to code.
- **Explain** what to do if you reject your hypothesis during debugging.
- **Explain** how multiple errors in the code can complicate debugging.

2 Second Reading

As always, we strongly recommend you sleep between your first and second reading. The second reading is where you want to focus on the higher-level learning objectives, which build on the lower-level learning objectives you worked on in the first reading. Sleeping gives your brain a chance to process the information from the first reading.

Note that many of the learning objectives in this section are going to require a lot of work and practice over the long term. You are not going to master them immediately. Instead, you should start to practice with them on the examples in this chapter. We have marked learning objectives with one star (*) if you are going to need significant practice, but should have fairly good mastery by the end of the assignments associated with this chapter. We have marked learning objectives with two stars (**) if we expect you to make a good start on them with reading and practice here, but that you will need to work on them well beyond this chapter.

- **Black Box Testing**
  - **Predict**** potential mistakes a programmer might make for a given programming problem.
  - **Develop**** a set of inputs which will show the incorrect behavior of a program if a given mistake is made.
  - **Formulate**** equivalence classes which effectively cover the behaviors of a program.

- **Determine**** edge and corner cases for a problem.
- **Assess**** when you need test cases at the boundary between two (or more) equivalence classes and **develop**** appropriate test cases.
- **Develop*** test cases for each error condition

- **White Box Testing**
  - **Compare and Contrast** black box and white box testing, and **illustrate*** how they can work together.
  - **Draw** the control flow graph for a given piece of code.
  - **Calculate** the coverage level provided by a set of test cases on a given piece of code.
  - **Construct*** a set of test cases which give a particular coverage level on a given piece of code.

- **Asserts**
  Invariants are at the boundary of what materials is covered in this class. We would like you to use asserts when you recognize the invariants of your algorithm. It is great to think carefully about invariants of your code, but that is not a learning objective we focus on here. If you take an Algorithms class, you should (ideally) have extensive coverage of invariants, and focus on learning objectives such as these:
  - **Determine**** the invariants of your algorithms.
  - **Convince**** yourself (or others) of how/why the invariants are maintained throughout the execution of your code.

  An Algorithms class may also involve formally proving that your code is correct! A proof is worth a thousand test cases ☺.

- **Testing and Types**
  - **Assess*** when the wrong number and/or types of input are appropriate/useful test cases.

- **Debugging**
  - **Compare and contrast**** inputs which work/do not work properly to **determine**** characteristics of failing inputs.
  - **Select*** between print statements and gdb as information gathering methods for a given situation.
  - **Use**** gdb as an information gathering and hypothesis testing tool for debugging.
    * Note: read Appendix D.2 for details, and practice with gdb!
  - **Formulate**** a good hypothesis for debugging.
  - **Assess**** how testable and actionable a hypothesis is for debugging.
  - **Revise**** a hypothesis to make it more testable and/or actionable.
  - **Select**** between constructing test cases, inspecting the internal state of the program, adding asserts, code inspection, or a combination of two or more of these techniques to test a hypothesis about your program.

- **Test**** a hypothesis about the problem with your program.
- **Assess**** whether your should modify existing code or re-design/re-write the code to fix a problem.
- **Assess**** when multiple errors are complicating debugging, and avoid confusing your debugging process.

3 Key Learning Objectives

This chapter covers two critical concepts: testing and debugging, which are going to take a LOT of practice over a long time to master. I want to underscore that I frequently talk to friends in industry about what students need to learn, and one of the most common things they say is "more about testing." Overall, your main learning objectives for this chapter are:

- **Formulate** a set of test cases for a problem which are *very likely* to catch bugs.
- **Assess** what test cases are missing based on test coverage, and **formulate** additional test cases to remedy those gaps.
- **Apply** the scientific method to **investigate** why a program does not work, then **revise** the code to fix the problem.

These are all very big, complex learning objectives that you will need to get good at quickly, but will spend a long time mastering. All of the learning objectives listed earlier feed into these three main skills.