

## Lab 2.1

# Tracking Down the Bugs

Chapter 7 (To Err is Human...) discusses strategies for debugging—finding and fixing problems with IT systems. In this lab, we focus on the early stages of debugging, where the objective is to precisely identify problems and begin to understand the reasons for the problems. Chapter 7 offered three example debugging scenarios: playing a videotape on a VCR, the butterfly web page, and printing. In this lab, you'll be working in another context, one that you will eventually become much more familiar with in later labs: programming. More specifically, you'll be provided with a web-based program, and your task will be to test it to determine whether it's working properly. As you practice the early stages of debugging in this context, you'll also be preparing yourself for writing and debugging your own computer programs later in this course.

### Vocabulary

All key vocabulary used in this lab are listed below, with closely related words listed together:

- specification
- black-box vs. white-box testing
- test case
- bug, error
- error reproducibility

### Post-Lab Questions

Write your answers after completing the main part of the lab:

1. What's the point of documenting procedures for reliably reproducing errors? Being able to reproduce an error reliably is important to make sure the error actually exists, but why is it valuable to have clearly documented procedures for reproducing bugs? How can this be helpful later in the debugging process?
  
2. Most of the bugs you will encounter in these labs are relatively easy to reproduce. Without too much testing, you can determine what sequence of actions you need to reliably cause the incorrect behavior on demand. With certain kinds of software, however, there are bugs that are very difficult to reproduce—incorrect program behavior that results from circumstances that are difficult to reproduce. Consider software like web browsers, e-mail programs, instant messaging clients (e.g., Yahoo Messenger, Trillian, MSN Messenger), and streaming media players (e.g., RealNetworks' player, WinAmp). What might make reliably reproducing

bugs with software like this difficult?

3. Consider the debugging VCR and HTML examples from Chapter 7. Do you consider these examples of black-box or white-box testing? Briefly explain your answers.
4. Choose one of the everyday engineered artifacts listed below and describe a procedure for testing that it works properly. We've intentionally chosen artifacts that are commonly understood, so we do not provide any specifications here. You don't need to describe how to test every single aspect of the artifact. Choose one or two more important aspects of the artifact.

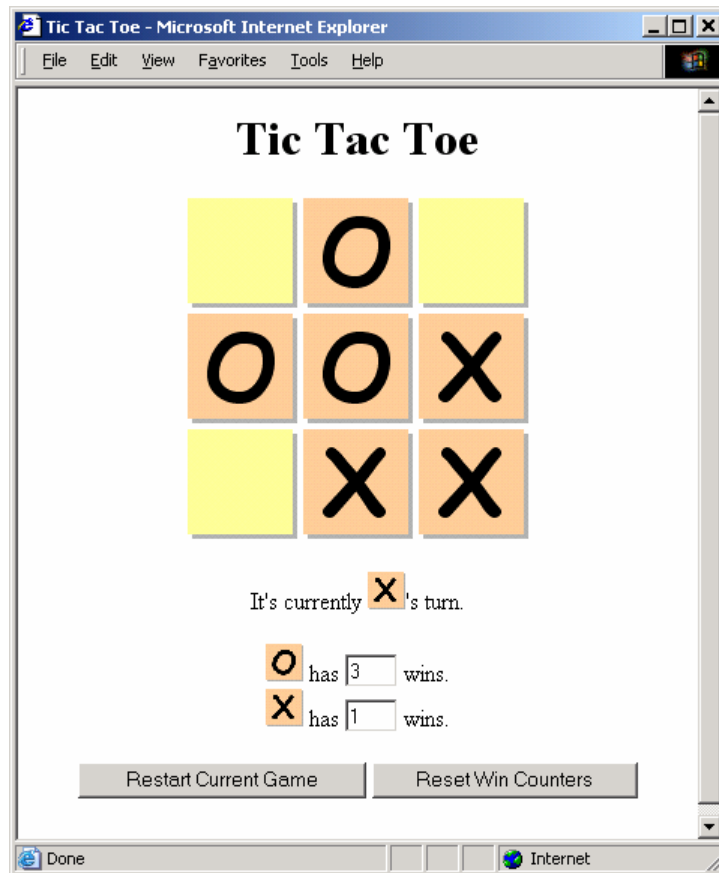
toaster, digital alarm clock, ATM, telephone (pushbutton and wired, not mobile)

## Discussion and Procedure

In this lab, you'll be working with a web-based, two-player tic-tac-toe game, pictured below. We'll describe precisely how the program is supposed to work, and your job will be to formulate and execute a plan for testing the program thoroughly. Your goal is to either verify that the program works exactly as it's specified or to find the bugs by identifying exactly how and when the program goes wrong.

**Quality Assurance Engineer.** There are professional engineers in software and hardware companies around the world whose job is similar to the work you will do in this lab. Quality assurance (QA) engineers are in a unique position, because they need to understand both the engineering that goes into products they test and the needs of the customers who will eventually be using those products. The work process for a QA engineer is much more extensive than the glimpse you get in this lab, but the basic problem is the same: make certain that a product works as it is specified by testing it according to a careful plan.

Before you get started with the lab, try playing with the page a little to familiarize yourself with the way it works. (Your instructor will tell you the URL for the game page.)



### Part 1. From to specification to tests

Since your goal is to identify what (if anything) works incorrectly with this program, we have to start by clearly and precisely defining what it means for this program to work correctly. Normally, we would refer to a *specification* for this information. A specification is a carefully written, formal description of how a system (a program, in this case) should work. In the interest of time, we'll provide a shorter, less formal specification for the tic-tac-toe program here, and based on it, you'll begin forming a plan for testing the program. As you read the specification, start thinking about what you would do (i.e., how you would use the program) to confirm that it works as specified.

The tic-tac-toe program lets two players (an O player and an X player) play a series of games of tic-tac-toe. The program keeps track of how many games each player has won, and clicking the “Reset Win Counters” sets both counters to zero. (When the page is first opened, these counters start at zero, as well.) Players take turns making moves by clicking on the tile they want to mark. Players may only mark an empty tile, and their turn ends when they mark a tile. The game ends when someone has won (three Os or Xs in a row) or when the board is full, which is a draw and counts as neither player's win.

The page displays whose turn it is at any given time. Players also alternate on who makes the first move. E.g., if O starts one game, X gets the first move of the next game. Clicking the Restart Current Game button clears the board and it becomes the turn of the player who started the game that was restarted.

Now let's take a closer look at parts of the specification and plan tests that would help us verify that the program works accordingly.

1. *For each of the specification excerpts below, describe how you would test that the program meets the specification. For each test, make sure to describe (1) what you would do with the program, as well as (2) how you expect the program to behave in response, noting the observations you will make to determine program correctness. A suggested test is given for the first excerpt as an example.*

**a. “The tic-tac-toe program lets two players (an O player and an X player) play a series of games of tic-tac-toe.”**

Play through several games, including at least one win by each player and one draw. Observe whether a new game properly starts after a game is completed.

**Imagination, attention to detail, and examined assumptions.** Before you proceed to the other specification excerpts, look at the sample test above and notice how a variety of different scenarios are tested (wins, draw). Try to think of all such variations in circumstances that might affect how the program runs. Debugging effectively by planning thorough tests requires quite a bit of attention to detail and imagination. It also requires you to recognize and examine your assumptions carefully.

- b. “The program keeps track of how many games each player has won...”**
  
- c. “Players take turns making moves...”**
  
- d. “Players may only mark an empty tile...”**
  
- e. “The game ends when someone has won (three Os or Xs in a row)...”**
  
- f. “The page displays whose turn it is at any given time.”**
  
- g. “Players also alternate on who makes the first move.”**
  
- h. “Clicking the Restart Current Game button clears the board and it becomes the turn of the player who started the game that was restarted.”**

Now you have a set of tests, each corresponding to part of the specification, that you can use to begin troubleshooting the program in a systematic, organized way.

## **Part 2. Testing and reporting bugs**

Carry out each of the tests you described above. Once you start testing, you might find that you need to adapt your testing instructions. Feel free to note any changes to the tests as you go.

For each test, in the space provided below, note whether you observed the behavior you expected. If not, you have most likely discovered a bug. Repeat the test as many times

as necessary to determine the steps you need to take to reliably *reproduce* the incorrect behavior to occur.

a.  pass /  fail

**incorrect behavior:**

**procedure to reproduce error:**

b.  pass /  fail

**incorrect behavior:**

**procedure to reproduce error:**

c.  pass /  fail

**incorrect behavior:**

**procedure to reproduce error:**

d.  pass /  fail

**incorrect behavior:**

**procedure to reproduce error:**

e.  pass /  fail

**incorrect behavior:**

**procedure to reproduce error:**

f.  pass /  fail

**incorrect behavior:**

**procedure to reproduce error:**

- g.  pass /  fail  
**incorrect behavior:**

**procedure to reproduce error:**

- h.  pass /  fail  
**incorrect behavior:**

**procedure to reproduce error:**

At this point, you've either convinced yourself that this program works according to specification, or you've identified one or more bugs and understand how to reproduce the problematic behavior. Normally, at this point, the debugging process would continue with attempting to fix the bugs, but without background in JavaScript programming, we'll stop here for now.

On the one hand, we've quite precisely identified buggy program behavior, but without being able to see or understand the program code, we're not sure exactly what needs to be fixed. This is a common limitation of the style of testing we did, which is called *black-box testing*. In black-box testing, you test a system without seeing “the inside”—its inner workings, the internal design. Instead, black-box testing relies on the externally observable behavior of a system.

In contrast, *white-box testing* is done with full knowledge of the system's internals. In other words, we not only know what the system does, we also know exactly how and why the system does what it does. Although white-box testing can identify certain bugs more easily than black-box testing, the trade-off is that white-box testing requires more technical expertise.

When you start programming (and debugging!) JavaScript in later labs, keep both of these strategies in mind. Especially if you are testing your own program, choosing black-box testing and distancing yourself from the internal design of the program can help you avoid making hasty assumptions about your program's correctness. On the other hand, white-box testing can be essential for pinpointing the exact reasons for a bug. The knowledge that white-box test results provides can be very valuable to the programmer in identifying and fixing bugs.