Lab 2.1
Tracking Down the Bugs

Chapter 6 (An Introduction to Debugging) discusses strategies for debugging—finding and fixing problems with IT systems. In this lab, you are given a program to test, and a specification for that program. The specification is just a careful statement of what the program *should* do. Your mission is to see if it does, and document any incorrect behaviours.

For the Instructor

Teaching Objectives

- *concepts*

  1. specification
  2. black-box vs. white-box testing
  3. error reproduction

- *skills*

  1. given a specification for a computer program, formulating a set of tests based directly on the specification
  2. carrying out such a set of planned tests
  3. recording test results and identifying bugs, documenting reproduction

---

For the Student

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

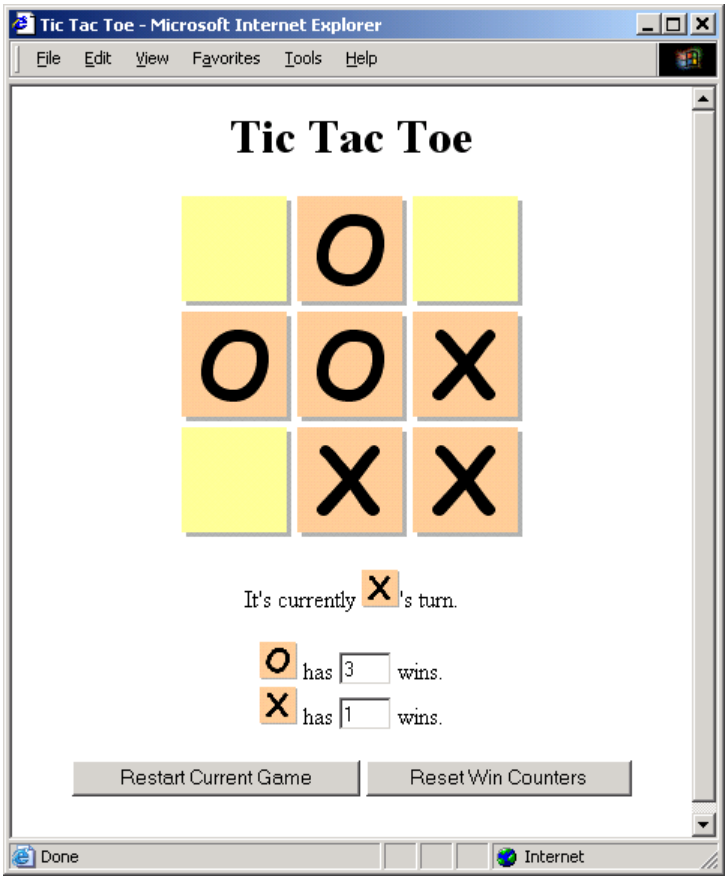
Discussion and Procedure

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. Bring up your favorite web browser and visit the game at `http://sandbox.mc.edu/~csc114/ttt_testing.html`. This program is written in Javascript, a language which is embedded in an HTML web page and is run by the browser.



Part 1. From 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.

Below, you will find the specification broken into parts. Each part describes some particular behavior of the program which can be tested. For each one, do the following three things:

1. Say what you would do to demonstrate that program actually has the specified behavior. Try to cover different situations to make sure the program behaves correctly consistently.

2. Say how you expect the program to behave under your testing.

3. State the the program behaves correctly, or say how it misbehaves. Be specific about when any misbehavior occurs, and what exactly the program does wrong. (Imagine you are explaining a problem with your car to a mechanic must figure out how to fix it.) Your description should be specific enough to allow the reader to reproduce the error on the first try.

Part a step 1 is done for you. Notice how it tries to cover the relevant conditions by making sure there is at least one win for each side, and one tie.

a. "The tic-tac-toe program lets two players (an O player and an X player) play <u>a series of games</u> of tic-tac-toe."

1. 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.

2.

3.

b. “The program keeps track of how many games each player has won…”

1.

2.

3.

c. “Players take turns making moves…”

1.

2.

3.

d. “Players may only mark an empty tile…”

1.

2.

3.

e. “The game ends when someone has won (three Os or Xs in a row)…”

1.

2.

3.

f. “The page displays whose turn it is at any given time.”

1.

2.

3.

g. “Players also alternate on who makes the first move.”

1.

2.

3.

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.”

1.

2.

3.

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.