Appendix

The simulation methods presented here are readily accessible to those familiar with programming. Each point on the Battleship grid is represented by a Boolean data type (i.e., a ship has been placed at that point or not) and referenced by a row/column pair in a two-dimensional array. Each type of ship is assigned a constant size. Starting at the upper left of the grid, the following procedure is followed to determine the number of possible ship configurations.

1. Place ship 1 at the next available space.
2. Place ship 2 at the next available space.
3. Place ship 3 at the next available space.
4. Place ship 4 at the next available space.
5. Place ship 5 at the next available space. A successful attempt counts as a possible configuration.
6. Repeat step 5 until all spaces have been attempted.
7. Remove ship 5 and begin at step 4.
8. Remove ships 4 and 5 and begin at step 3.
9. Remove ships 3–5 and begin at step 2.
10. Remove ships 2–5 and begin at step 1.

Each attempt tries to insert a ship both horizontally (i.e., to the right) and vertically (i.e., down) at each grid point. These steps are most easily performed using a recursive algorithm. In pseudocode, this looks like the following:

```
main() {
  insertShip(all rows, all columns, ship #1, new grid);
}

insertShip(row, column, ship, grid) {

  if checkHorizontal(row, column, ship, grid) {  // check if possible
    fillHorizontal(row, column, ship, grid);
    if (all five ships have been placed) addSuccessfulGrid(grid);
    else insertShip(all rows, all columns, next ship, new grid);
  }

  if checkVertical(row, column, ship, grid) {  // check if possible
    fillVertical(row, column, ship, grid);
    if (all five ships have been placed) addSuccessfulGrid(grid);
    else insertShip(all rows, all columns, next ship, new grid);
  }
}
```

To simulate several shots, the program must keep track of both the type (i.e., hit or miss) and location of all shots. This data is then fed into a new simulation (i.e., the next shot) which accounts for previous hits and misses as 100% or 0% probability that a ship exists at a certain point.

With the huge numbers of possible configurations presented here, one may ask how I can be sure that my program is working correctly. After all, no one can say, "Of course there are 5.8 billion ways to arrange these ships on a 10 × 10 grid with at least one space in between them; isn't it obvious?" The following constituted my error checking:

1. Simulations of 3 × 3 grids with smaller ships that could be checked by hand.
2. The results are symmetric about many rotation axes and reflection planes, as expected.
3. Each simulation carries with it an output file that displays grid-point probabilities. With spaces between ships, the worst-possible-guessing path seen in **Error! Reference source not found.** should converge to one possibility (and its rotations and reflections) with two configurations (i.e., switching the submarine and destroyer). In addition, there are

always even numbers of possible configurations because of these two ships. Simulations confirmed both of these facts.

4. With no spaces between ships, the worst possible configuration is piling ships into a corner. Regardless of the grid size, each simulation provided the same ship pattern having 14 possible configurations (I'll leave it to the reader to show this is true), seen in Figure 1.

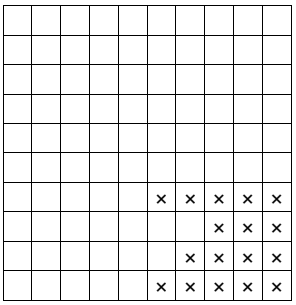

Figure 1. The worst possible way to arrange your ships with no spaces between them.

5. Each simulation step's methodology can be observed and tested through output files by retracing the patterns of placing ships.
6. The results match intuition—namely, taking shots in spiral patterns and avoiding placing your ships on the edges.
7. Of course there are 5.8 billion ways to arrange these ships on a 10 × 10 grid with at least one space in between them; isn't it obvious?