1. What are you trying to model? Include a brief description that would give someone unfamiliar with the topic a basic understanding of your goal.

   a. We are trying to model traces for a player in the board game Battleship. In Battleship, there are two players with separate nxn grids, neither of whom can see the other player's board. Each player places boats on their board of length >= 1, and then takes turns guessing the locations of the other player's boats. Once a player guesses correctly all the (row, col) locations of their opponent's boats, they have sunk their opponent's fleet and won the game!

      We want to model properties that should always hold for a valid game of Battleship, like:
      - What does it mean for a board to be well formed?
      - What types of moves are possible between two boards?
      - When does a player lose?

2. Overview of:

   a. Your model design choices. At a high level, what do each of your sigs and preds represent in the context of the model? Justify the purpose for their existence and how they fit together.

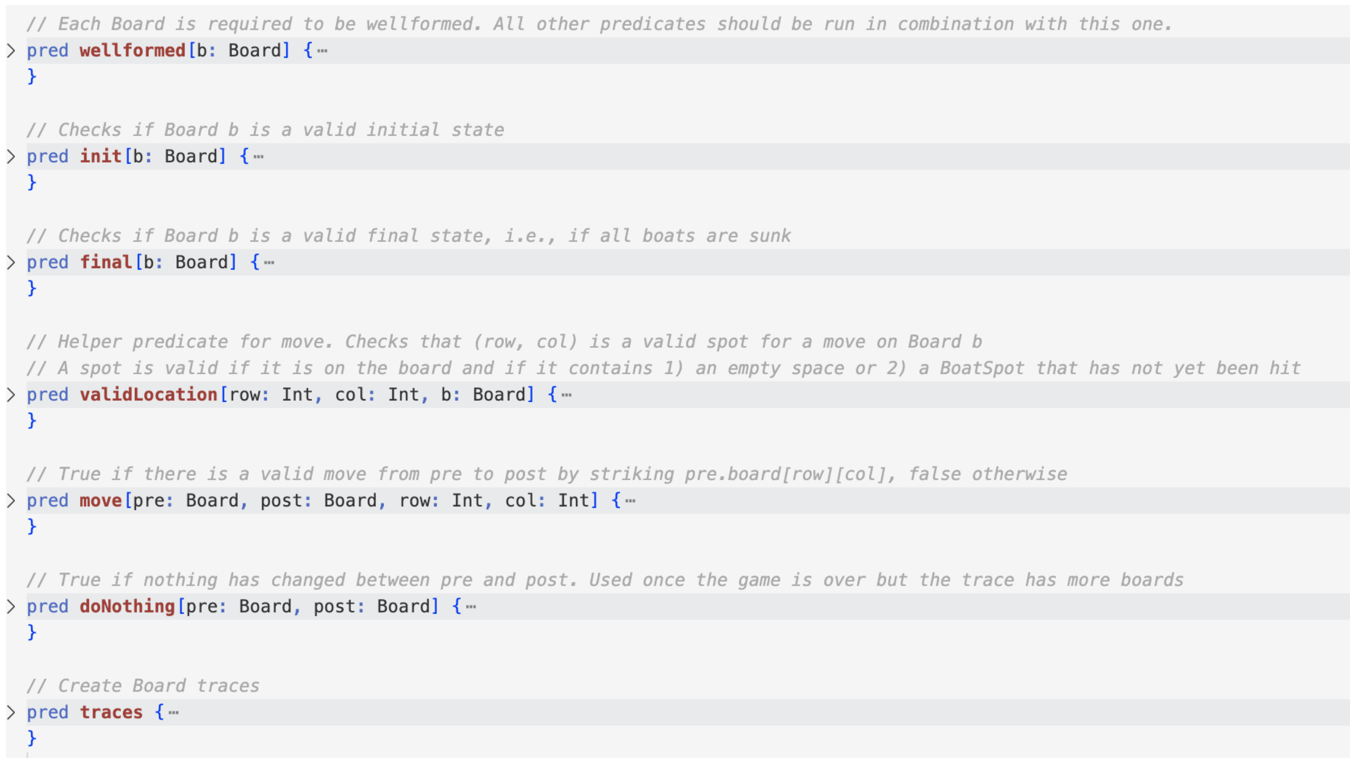
      We model the game from one player's perspective, as their board gets hit by the other player. The board contains three boats, each of length 2.
      We have the following sigs:

```
abstract sig Boolean {}
one sig True, False extends Boolean {}

// Links the Boards together
one sig Game {
  initial: one Board,
  next: pfunc Board -> Board // next Board in the trace
}

// sig for cells on the Board
abstract sig Space {}

// A cell on the Board that contains 1/2 of a Boat
sig BoatSpot extends Space {}
// A cell on the Board where the player tried to strike a BoatSpot, but missed
sig MissedStrike extends Space {}

// note that there is no "Empty" sig--we made this modeling choice because it takes
less storage to just write "no b.board[row][col]" for some row, col : Int, Board b
than to have a bunch of Empty objects

// Boats are comprised of two adjacent BoatSpots
sig Boat {
   spot1: one BoatSpot,
   spot2: one BoatSpot
}

// Represents the state of the game
sig Board {
   board: pfunc Int -> Int -> Space, // row index -> column index -> Space
   boats: set Boat, // set of boats that belong to this Board
   hit_boatspots: func BoatSpot -> Boolean, // for each BoatSpot, maps to True if it
has been hit in this state, false otherwise
}
```

We use the Game sig to create traces of multiple Boards, with a linear relationship between a Board and the next state it reaches by executing one move. Boards represent a state of the game, with a board, boats, a map of all the boat cells and whether they're been hit in this state, and a boolean representing whether they've lost.

A Boat is made up of two BoatSpots, which are an extension of the Space sig. A MissedStrike is a Space where the opponent tried to hit a boat on the board, but missed.

We have the following predicates:



```
  // Each Board is required to be wellformed. All other predicates should be run in combination with this one.
> pred wellformed[b: Board] {⋯
  }

  // Checks if Board b is a valid initial state
> pred init[b: Board] {⋯
  }

  // Checks if Board b is a valid final state, i.e., if all boats are sunk
> pred final[b: Board] {⋯
  }

  // Helper predicate for move. Checks that (row, col) is a valid spot for a move on Board b
  // A spot is valid if it is on the board and if it contains 1) an empty space or 2) a BoatSpot that has not yet been hit
> pred validLocation[row: Int, col: Int, b: Board] {⋯
  }

  // True if there is a valid move from pre to post by striking pre.board[row][col], false otherwise
> pred move[pre: Board, post: Board, row: Int, col: Int] {⋯
  }

  // True if nothing has changed between pre and post. Used once the game is over but the trace has more boards
> pred doNothing[pre: Board, post: Board] {⋯
  }

  // Create Board traces
> pred traces {⋯
  }
```

wellformed should be run in combination with all these predicates, because a board should always be wellformed if we're testing other properties about it. Traces should start at an initial state, and for each board and its next in a trace, we try move between those boards, and if and only if the board is in a final state should we doNothing.

b. What checks or run statements you wrote
    i. We wrote run statements at the bottom of battleship.frg to run each of our predicates. Those run statements are commented with descriptions of what each do and what properties you can check in the evaluator.
    ii. The tests for our model are in battleship.tests.frg. We included test suites for each predicate with passing and failing examples for each. We also wrote test expects for properties that we would expect to be satisfiable/unsatisfiable/theorem for our model. For more details, please see the comments in battleship.tests.frg.

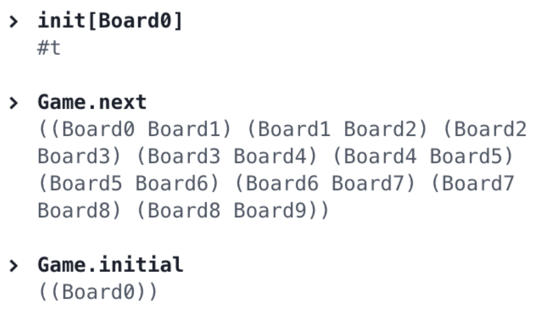
c. What we should expect to see from an instance produced by the Sterling visualizer. How should we look at and interpret an instance created by your spec? Did you create a custom visualization, or did you use the default?

    We used the default visualizer. We'll walk through an example of what Sterling produces after running the last run statement we include in battleship.frg:

```
run {
   all b : Board | wellformed[b]
   traces
} for exactly 10 Board, exactly 3 Boat, exactly 6 BoatSpot, exactly 3 MissedStrike for
{next is linear}
```

    We found that the Evaluator was the most helpful for looking at instances, with the Table view second-most helpful and the Graph view least helpful. We'll go through this example in the Evaluator.

    First, let's check properties about `Game.initial` and `Game.next`:



```
> init[Board0]
  #t

> Game.next
  ((Board0 Board1) (Board1 Board2) (Board2
  Board3) (Board3 Board4) (Board4 Board5)
  (Board5 Board6) (Board6 Board7) (Board7
  Board8) (Board8 Board9))

> Game.initial
  ((Board0))
```

    We've verified that `Game.initial` is a valid initial state, and we can see the linear trace of the 10 Board objects.

Now, we can check how we move (or doNothing) between each of these Boards. There's two main fields we want to check for a given board b: b.board and b.hit_boatspots. Let's see the output for Board0:

```
> Board0.hit_boatspots
  ((BoatSpot0 False0) (BoatSpot1 False0)
  (BoatSpot2 False0) (BoatSpot3 False0)
  (BoatSpot4 False0) (BoatSpot5 False0))

> Board0.board
  ((2 4 BoatSpot0) (3 1 BoatSpot1) (3 4
  BoatSpot3) (4 1 BoatSpot4) (4 3 BoatSpot2)
  (4 4 BoatSpot5))
```

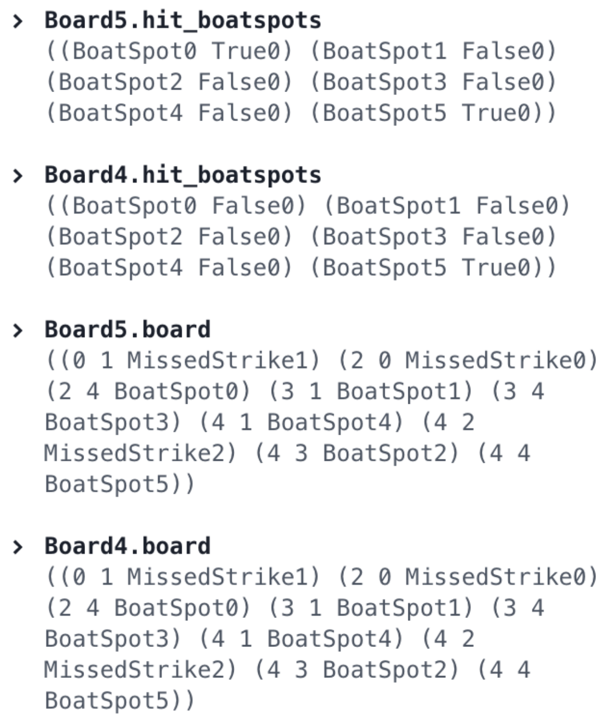
Board0.hit_boatspots tells us whether each of the six Boatspots is hit on this board. In this case, they all map to False, which means none of the BoatSpots have been hit.
Board0.board gives us the layout of the board. For example, BoatSpot0 lives at Board0.board[2][4]. Once a Space object is on a Board, it will stay at the same row, column throughout traces (moving your boats or MissedStrikes mid-game would be cheating!)

Now we want to validate properties about the trace. Since Board0 is an initial state, we should be able execute a move. Let's verify that a move occured between Board0 and Board1:

```
> move[Board0, Board1, 2, 0]
  #t

> Board1.hit_boatspots
  ((BoatSpot0 False0) (BoatSpot1 False0)
  (BoatSpot2 False0) (BoatSpot3 False0)
  (BoatSpot4 False0) (BoatSpot5 False0))

> Board1.board
  ((2 0 MissedStrike0) (2 4 BoatSpot0) (3 1
  BoatSpot1) (3 4 BoatSpot3) (4 1 BoatSpot4)
  (4 3 BoatSpot2) (4 4 BoatSpot5))
```

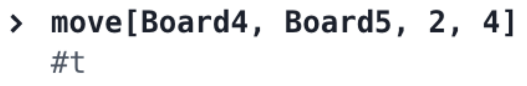
We see that we didn't hit any BoatSpots between Board0 and Board1, but we have a new MissedStrike at a (2, 0), which we do not see as a (row, col) location on Board0.board. This means that Board0[2][0] was an empty space, so we made a move that missed from Board0 to Board1. To check this hypothesis, we run move[Board0, Board1, 2, 0] and see it's true, as expected.

Let's pick two boards in the middle of the trace, Board4 and Board5:



```
> Board5.hit_boatspots
  ((BoatSpot0 True0) (BoatSpot1 False0)
  (BoatSpot2 False0) (BoatSpot3 False0)
  (BoatSpot4 False0) (BoatSpot5 True0))

> Board4.hit_boatspots
  ((BoatSpot0 False0) (BoatSpot1 False0)
  (BoatSpot2 False0) (BoatSpot3 False0)
  (BoatSpot4 False0) (BoatSpot5 True0))

> Board5.board
  ((0 1 MissedStrike1) (2 0 MissedStrike0)
  (2 4 BoatSpot0) (3 1 BoatSpot1) (3 4
  BoatSpot3) (4 1 BoatSpot4) (4 2
  MissedStrike2) (4 3 BoatSpot2) (4 4
  BoatSpot5))

> Board4.board
  ((0 1 MissedStrike1) (2 0 MissedStrike0)
  (2 4 BoatSpot0) (3 1 BoatSpot1) (3 4
  BoatSpot3) (4 1 BoatSpot4) (4 2
  MissedStrike2) (4 3 BoatSpot2) (4 4
  BoatSpot5))
```

First, we observe that Board5.board = Board4.Board. This indicates that we either hit a BoatSpot or doNothing ran. To see which, we print hit_boatspots, which indicates that we hit BoatSpot0, which means we must have run move. To verify that move is true for these coordinates, we run:



```
> move[Board4, Board5, 2, 4]
  #t
```