Case study: The Game of Vexil

Before we delve deeply into processes, messages, and tasks, I want to introduce a simple game (Vexil) to help motivate our study. This is strictly a work in progress, and it's a game of my own creation; so bear with me.

You might ask: Is this worth the trouble? I believe it is. Although the context is simple and contrived, it offers enough complexity to be interesting, and it affords us plenty of opportunities for iterative development and learning new concepts.

I'll begin by explaining the game and the concepts behind it. We will look at a simple first approximation, and then we'll enahnce it and move toward real parallelization. We won't finish in this chapter, because we'll need concepts that have not yet been introduced in this book.

Let's begin with a description of the game, and then look at a simple implementation in Ruby. I should stress again, of course, that this book is **not** about Ruby. We're **reading** the Ruby code to facilitate **writing** the Elixir code, moving from the more familiar to the less familiar.

Vexil and its Heritage

This is a simple "capture the flag" type of game. I call it Vexil (a name derived from the Latin **vexillum** for "flag").

Vexil is **not** a board game, although glancing at it might make you think so. It actually derives from (**discuss Core Wars and Darwin**).

In a board game like chess or checkers, there are two opponents, each with absolute knowledge of the entire board. But in Vexil, the opponents are more like **teams** (labeled "red" and "blue"). Each player on a team is intended to act autonomously, with no global knowledge of the grid and no single point of control.

So a good analogy is the "battling bots" type of game which we've seen many times in the past. I'm sure you can see the direction this is going. Each player or piece will (ultimately) be controlled by a single process; these processes will collaborate to defeat those on the other team. As such, it will finally be a battle of algorithms, where coders write their best logic for the bots and then turn them loose on the grid. The "referee" process will manage communication, enforce the rules, prevent (easy) cheating, and declare a winner.

So on to the details. The Vexil grid is 21 by 21 for a total of 441 cells. The Red team originates in the lower left portion, diagonally across from the Blue team on the upper right.

Each team views the grid in its own coordinate system. The **x** and **y** values can vary from 1 to 21. For example, the cell that the Red team calls `(3,5)` will be viewed as `(19,17)` from the Blue side.

Each team has a "flag" that is randomly placed by the referee within four cells of the corner (i.e., somewhere in that 4-by-4 area). We'll call this zone 1. Besides the flag, nothing starts out in this area,

There are three kinds of pieces or players. Each kind is characterized by its abilities in several areas of behavior:

- It can **see** a square sub-grid centered on itself and "know" what is in each of the nearby occupied cells (friend or for or flag);
- It can **move** a certain number of cells per turn, any combination of horizontal and vertical moves;
- It can **defend** itself by withstanding an attack up to a certain number of points of damage;
- It can **attack** and inflict damage points on an opposing piece;
- It can attack within a limited **range** (true distance between cells),

The following rules apply. Some are dictated by common sense, while others are more or less arbitrary.

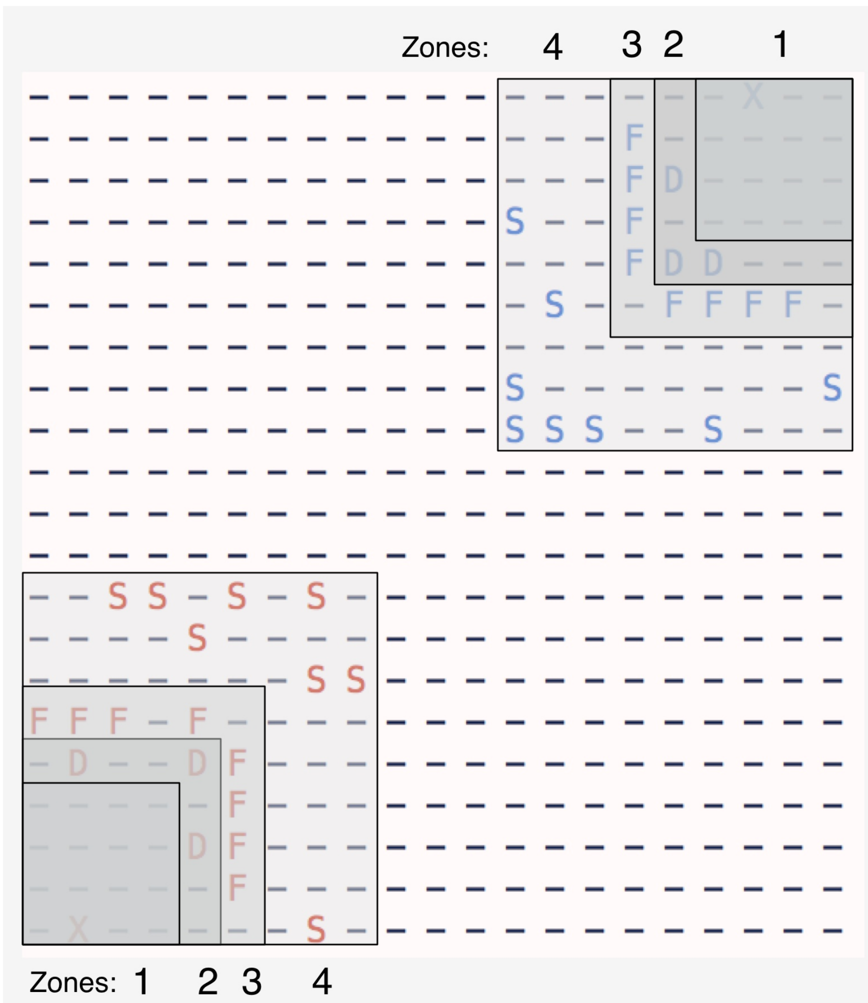
- No cell can contain more than one piece at a time; possible collisions will be resolved randomly.
- Every piece "knows" where its team's flag is.
- No piece knows the enemy flag's location until it is in visual range.
- A piece cannot "see" through other pieces regardless of range.
- A piece can always see farther than it can move.
- A piece always has an attack range less than its range of motion.
- Pieces may communicate with their own team members (by "radio") regardless of distance.
- Mutual attacks will be resolved randomly.
- When a piece receives damage, it never recuperates; when its constitution (or "hit points") reaches zero, it dies and is removed from the grid.

- Pieces never run out of "ammunition" (ability to attack).
- Pieces may not attack their own team.
- When a piece moves onto the cell containing the enemy's flag, the game is over.
- A piece may of course not capture its own flag.

So as I said, there are three kinds of pieces. The **defender** cannot see or move very far, but it can attack and it is difficult to kill. The **scout** can see far and move quickly, but cannot attack (or withstand attacks) very well. The **fighter** is faster than the defender but slower than the scout; it is tougher than the scout, but not so tough as the defender; and it is the best attacker of all. This information is summarized in this table:

	Can move	Can see	Defending	Attacking	Range
Defender	2	3	6	4	2
Fighter	4	6	6	6	4
Scout	5	8	3	2	1

For those of us who are visually oriented, here is a diagram of the grid:



Zone 2 is an L-shaped area consisting of the 5-by-5 area nearest the corner, **minus** the cells in zone 1. Here the referee randomly places three defenders.

Zones 3 and 4 are also L-shaped areas. The referee randomly populates zone 3 with six fighters and zone 4 with six scouts.

A First Approximation in Ruby

I don't usually include really large, complete pieces of code in a book. In this book, I am making an exception.

For a little more digestibility, this game is split into multiple files. Let's look first at a "roadmap" of these files and the classes and methods they define.

```
# File: grid.rb
  class Grid
    def initialize
    def [](team, x, y)
    def []=(team, x, y, obj)
    def coordinates(team, x, y)

# File: misc.rb
  class String
    def red
    def blue

# File: referee.rb
  class Referee
    def initialize
    def show_cell(xx, yy)
    def record(line)
    def display
    def [](team, x, y)
    def setup
    def turn
    def pause
    def move(team, x0, y0, x1, y1)
    def attack(qty, team, x, y)
    def place(team, kind, x, y)
    def over!
    def over?

# File: pieces.rb

  class Bot
    def initialize(team, data, x, y)  #
    def to_s
    def who
    def move(dx, dy)
    def move!(dx, dy)
    def where
    def enemy?(piece)
    def within(n)
    def can_see
    def can_attack
    def turn
    def attack(qty, team, x, y)

  class Defender < Bot
    def initialize(team, x, y)
    def turn

  class Fighter < Bot
    def initialize(team, x, y)
    def turn

  class Scout < Bot
    def initialize(team, x, y)
    def turn

  class Flag < Bot
    def initialize(team, x, y)
```

The `Grid` class handles the logic of the "board" or "field" on which the pieces move. Coordinates are in **x-y** form and are relative to each team's corner. The "absolute" coordinates are the red one (origin lower left). The coordinates are 1-based (ranging from 1 to 21).

The `Referee` class handles all the details of the game itself in an impartial way. For example, when a piece attacks another, the `attack` method in `Referee` manages it, recording damage and removing dead pieces from the grid.

The `display` method shows the grid and ongoing history of the game on the terminal; it currently works in a very "dumb" way by clearing the screen and redrawing the contents. The pieces are colored via ANSI terninal codes; refer to the reopened `String` class with methods `red` and `blue` added.

The `setup` method will set up all the pieces on the grid. It does so by calling `place` repearedly (which supports some degree of pseudorandnomness).

The `record` method simply records the moves made by the bots on either side. The `over?` method returns a Boolean value telling whether the game is finished or not; by contrast, the `over!` method **declares** the game to be finished.

The `Bot` class is naturally one of the most important. Every kind of piece inherits from this class (even the flags, which is arguably a bad design decision).

Some methods such as `to_s`, `who`, and `where` are mere convenience methods. Other methods are informational, such as `enemy?` which determines whether another piece is friend or foe, and `within` which returns a list of other pieces within a certain range. (Note that `within` doesn't use a true Cartesian distance; it looks in a square centered on the bot in question.)

The `can_see` and `can_attack` methods simply let a bot know which of its neighboring enemies it can see and/or attack. Naturally `within` is used here.

The `turn` method is the "lifecycle" of a bot, its **raison d'etre**. Fighters are aggressive, scouts are curious but cowardly, and defenders are powerful and armored but very stay-at-home types.

When a bot takes a turn, its behavior is given in terms of "looking around" and then moving and/or attacking. The `move` method requests that the `Referee` move the bot to that location; this request may be denied if, for example, another bot moves there first. Thus the referee acts to prevent race conditions by serializing access to the grid. The `move!` method will try one move after another, in the "general" direction intended, until one of them succeeds. It is possible that **none** of these will succeed.

Finally, `attack` will "hit" an enemy with some kind of unspecified weapon. The strength of the attack determines the damage to the enemy; if the enemy's strength is exhausted, it will "die."

Here is a screenshot of a game in progress. If you are reading the print edition of this book, I am sorry to note that it induces temporary color blindness.

```
- - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - X
- - - - - - - - - - - - - s - - d - - - -
- - - - - - - - - - - - - - - - s - - - -
- - - - - - - - - - - - - - - - d - - d -
- - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - f - - - - - - - - - s -
- - - - - - - - - - - - - f - - - - - - -
- - - - - - - - - - - - - - - - - - - - -
- - - - - - - f - - - - - - - - - - - - -
- - - - - - s - - - f f - - - - - - - - -
- - - s - - - - - - f - - f - - - - - - -
- - - - - - - - - - - - f - - - - - - - -
- - - - - - s - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - -
- - d - d - - - - - - - - - - - - - - - -
- - - - - s - - - - s - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - -
s - - - d s - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - -

155. Fighter @ 7,12  attacks Fighter @ 12,11
156. Fighter @ 7,12  attacks Fighter @ 11,9
157. Fighter @ 12,11 attacks Fighter @ 12,9
158. Fighter @ 12,11 attacks Fighter @ 11,12
159. Fighter @ 12,11 attacks Fighter @ 10,12
160. Fighter @ 12,11 attacks Fighter @ 9,14
161. Fighter @ 12,11 attacks Fighter @ 8,13
162. Fighter @ 12,9  attacks Fighter @ 14,13
163. Fighter @ 12,9  attacks Fighter @ 11,9
164. Fighter @ 12,9  attacks Fighter @ 10,14
165. Scout @ 17,15 can see enemy flag at 21,20
166. Scout @ 18,17 can see enemy flag at 21,20
167. Scout @ 18,17 attacks Defender @ 5,5
168. Scout @ 18,17 attacks Defender @ 3,5
169. blue wins!
```

That's an overview of how this game works. Read the Ruby code, try it out, and then let's look at how we will do (roughly) the same thing in Elixir.

An Implementation in Elixir

The overall design will be similar in spirit to the Ruby version. Of course, merely changing from object-oriented thinking to functional programming will introduce some differences.

`Referee` and `Bot` are now modules rather than classes. There is a `Bot` struct as well.

The "main program" is very simple. I've left it as a `.exs` file.

```elixir
:rand.seed(:exsplus, {0,0,0})    # Remove later

game = Referee.new
Referee.display(game)
game = Referee.start(game)

:timer.sleep 40000     # Fix later
```

Once again, you can think of the referee as being the "gatekeeper" of the game, controlling access to the grid. In a sense it **is** the game, as you may notice from some ambiguity in my choice of names.

The referee is modeled as a process, and so are all of the pieces. Think of the referee as a server and the bots as clients. (This will become more explicit after we look more at `GenServer` and related topics.)

The `Referee` module:

```elixir
defmodule Referee do

  defstruct [:grid, :bots, :pid, :over?]

  def new do
    {grid, bots} = setup(%{})
    game = %Referee{grid: grid, bots: bots, pid: nil, over?: false}
    bot = Bot.defender(:red, 3, 5)
    list = Bot.within(game, bot, 2)
    _ =list
    game
  end

  def verify(where, sig1, sig2) do
    _ = where
#   IO.puts "#{where}: #{Base.encode16(sig1)} -> #{Base.encode16(sig2)}"
    if sig1 == sig2 do
      IO.puts "  grid has NOT changed"
    else
      IO.puts "  grid has changed"
    end
  end

  def place(grid, bots, team, kind, x, y) do
    x2 = if Range.range?(x), do: rand(x), else: x
    y2 = if Range.range?(y), do: rand(y), else: y

    bot = Bot.make(kind, team, x2, y2)
    {grid, bots} =
    if Grid.cell_empty?(grid, {team, x2, y2}) do
      {Grid.put(grid, {team, x2, y2}, bot), bots}
    else
      place(grid, bots, team, kind, x, y)
    end
    bots = [bot] ++ bots
    {grid, bots}
  end

  def rand(n) when is_integer(n), do: :rand.uniform(n)
  def rand(n1..n2), do: :rand.uniform(n2 - n1 + 1) + n1 - 1

  def setup(grid) do
    {grid, redbots} = setup(grid, :red)
    {grid, bluebots} = setup(grid, :blue)
    bots = redbots ++ bluebots
    {grid, bots}
  end

  def setup(grid, team) do
    bots = []
    {grid, bots} = place(grid, bots, team, :defender, 5, 5)
    {grid, bots} = place(grid, bots, team, :defender, 5, 1..4)
    {grid, bots} = place(grid, bots, team, :defender, 1..4, 5)

    {grid, bots} = place(grid, bots, team, :fighter, 6, 1..5)
    {grid, bots} = place(grid, bots, team, :fighter, 6, 1..5)
    {grid, bots} = place(grid, bots, team, :fighter, 6, 1..5)
    {grid, bots} = place(grid, bots, team, :fighter, 6, 1..5)
    {grid, bots} = place(grid, bots, team, :fighter, 1..5, 6)
    {grid, bots} = place(grid, bots, team, :fighter, 1..5, 6)
    {grid, bots} = place(grid, bots, team, :fighter, 1..5, 6)
    {grid, bots} = place(grid, bots, team, :fighter, 1..5, 6)

    {grid, bots} = place(grid, bots, team, :scout, 8..9, 1..9)
    {grid, bots} = place(grid, bots, team, :scout, 8..9, 1..9)
    {grid, bots} = place(grid, bots, team, :scout, 8..9, 1..9)
    {grid, bots} = place(grid, bots, team, :scout, 8..9, 1..9)
    {grid, bots} = place(grid, bots, team, :scout, 1..9, 8..9)
    {grid, bots} = place(grid, bots, team, :scout, 1..9, 8..9)
    {grid, bots} = place(grid, bots, team, :scout, 1..9, 8..9)
    {grid, bots} = place(grid, bots, team, :scout, 1..9, 8..9)

    {grid, bots} = place(grid, bots, team, :flag, 1..4, 1..4)

    {grid, bots}
```

```
    end

    def display(game) do
      Grid.display(game.grid)
    end

    def move(game, team, x0, y0, x1, y1) do
      grid = game.grid
      piece = Grid.get(grid, {team, x0, y0})
      dest = Grid.get(grid, {team, x1, y1})
      {grid, ret} =
        cond do
          dest == nil ->
  #         IO.puts "move: normal case"
            g = Grid.put(grid, {team, x1, y1}, piece)
            g = Grid.put(g, {team, x0, y0}, nil)
            {g, true}
          dest in [:redflag, :blueflag] ->
            g = Grid.put(grid, {team, x1, y1}, piece)
            g = Grid.put(g, {team, x0, y0}, nil)
            # FIXME mark game as over
            IO.puts "case 2: Moved onto #{dest} - game over - FIXME"
            {g, false}  # logic??
          true ->
            IO.puts "case 3: SOMETHING WRONG? Can't move #{inspect piece} onto #{inspect dest}"
            {grid, false}
        end
      game = %Referee{game | grid: grid}
      display(game)

      {game, ret}
    end

    def over?(game), do: game.over?

    def record(_x, _y, _z), do: nil  # FIXME

    def start(game) do
      pid = spawn Referee, :mainloop, [game]
      game = %Referee{game | pid: pid}
      Enum.each(game.bots, fn(bot) -> Bot.awaken(bot, game) end)
      game
    end

    def mainloop(game) do
      g = receive do
        {caller, _bot_game, :move, team, x0, y0, x1, y1} ->
  #       IO.puts "mainloop: #{team} moves from #{inspect {x0, y0}} to #{inspect {x1, y1}}"
          {g2, ret} = move(game, team, x0, y0, x1, y1)
          if ret do
            send(caller, {g2, ret})
          end
          g2
      end
  IO.puts "cp1"
      display(g)
  IO.puts "cp2"
      :timer.sleep 200
      mainloop(g) # tail call optimized
    end

  end
```

The `Bots` module:

```
defmodule Bot do

  defstruct team: nil, kind: nil, move: nil, see: nil,
            defend: nil, attack: nil, range: nil,
            x: nil, y: nil

  def defender(team, x, y) do
    %Bot{team: team, kind: :defender, move: 2, see: 3, defend: 6, attack: 4, range: 2, x: x, y: y}
  end

  def fighter(team, x, y) do
```

```
      %Bot{team: team, kind: :fighter, move: 2, see: 3, defend: 6, attack: 4, range: 2, x: x, y: y} # FIXME
    end

    def scout(team, x, y) do
      %Bot{team: team, kind: :scout, move: 2, see: 3, defend: 6, attack: 4, range: 2, x: x, y: y} # FIXME
    end

    def flag(team, x, y) do
      %Bot{team: team, kind: :flag, move: 2, see: 3, defend: 6, attack: 4, range: 2, x: x, y: y} # FIXME
    end

    def make(kind, team, x, y) do
      apply(Bot, kind, [team, x, y])
    end

    def to_string(bot) do
      initial = bot.kind |> Atom.to_string |> String.capitalize |> String.first
      char = if bot.kind == :flag, do: "X", else: initial
      str =
      if bot.team == :red do
        "\e[31m#{char}\e[0m"
      else
        "\e[34m#{char}\e[0m"
      end
      str
    end

  # Move to Grid??

    def within(game, bot, n) do
      found = []
      grid = game.grid
      team = bot.team
      {x, y} = {bot.x, bot.y}
      {x0, y0, x1, y1} = {x-n, x+n, y-n, y+n}
      {xr, yr} = {x0..x1, y0..y1}

      filter = &(&1 == nil or Bot.where(&1) == Bot.where(bot))
      list = for x <- xr, y <- yr do
        piece = Grid.get(grid, {team, x, y})
      end
      Enum.reject(list, filter)
    end

    def where(bot) do
      {bot.x, bot.y}
    end

    def enemy?(me, piece) do
      me.team != piece.team
    end

    def can_see(game, me) do
      within(game, me, me.see)
    end

    def can_attack(game, me) do  # Things I can attack
      list = within(game, me, me.range)
  #   list = list.reject {|x| x.is_a? Flag }
    end

    def seek_flag(game, me) do  # FIXME!!
      stuff = can_see(game, me)
  # Ruby code:
  #   flag = stuff.select {|x| x.is_a? Flag }.first
  #   unless flag.nil?  # Remember to tell others where flag is
  #     fx, fy = flag.where
  #     fx, fy = 22 - fx, 22 - fy  # native coordinates
  #     dx, dy = fx - @x, fy - @y
  #     $game.record "#{self.who} can see enemy flag at #{fx},#{fy}"
  #     if (dx.abs + dy.abs) <= @move  # we can get there
  #       $game.record "#{self.who} captures flag!"
  #       move(dx, dy)
  #     end
  #   end
    end

```

```
  def move(game, true, bot, dx, dy), do: {game, bot, false}

  def move(game, false, bot, dx, dy) do
    x2 = bot.x + dx
    y2 = bot.y + dy

    # send msg to referee
    {g, result} = Comms.sendrecv(game.pid, {self(), game, :move, bot.team, bot.x, bot.y, x2, y2})
    bot2 = if result do
      Referee.record(g, :move, bot)  # $game.record("#{self.who} moves to #@x,#@y")
      b2 = %Bot{bot | x: x2}
      b3 = %Bot{b2  | y: y2}
    else
      bot
    end

    {game, bot, result}
  end

  def try_moves(game, bot, dx, dy) do
    deltas = [{dx, dy}, {dx-1, dy+1}, {dx+1, dy-1}, {dx-2, dy+2}, {dx+2, dy-2}]
    {game, bot} = attempt_move(game, bot, deltas)
    {game, bot}
  end

  def attempt_move(game, bot, []), do: {game, bot}

  def attempt_move(game, bot, [dest | rest]) do
    {dx, dy} = dest
    {game, bot, result} = move(game, Referee.over?(game), bot, dx, dy)
    if result do
      {game, bot}
    else
      attempt_move(game, bot, rest)
    end
  end

## credit mononym

  def turn(:fighter, bot, game) do
    # FIXME will call move, attack
    {game, bot} = try_moves(game, bot, 2, 2)
##    seek_flag
##
##    @strength = @attack
##    victims = can_attack
##    victims.each {|enemy| try_attack(2, enemy) || break }
##    move!(2, 2)
    {game, bot}
  end

  def turn(:defender, bot, game) do
    # FIXME will call move, attack
##    @strength = @attack
##    victims = can_attack
##    victims.each {|enemy| try_attack(3, enemy) || break }
    {game, bot}
  end

  def turn(:scout, bot, game) do
    # FIXME will call move, attack
    try_moves(game, bot, 3, 3)
##    seek_flag
##
##    @strength = @attack
##    victims = can_attack
##    victims.each {|enemy| try_attack(1, enemy) || break }
##    move!(3, 3)
    {game, bot}
  end

  def turn(:flag, bot, game), do: {game, bot}

  def mainloop(bot, game) do
    # the bot lives its life -- run, attack, whatever
    # see 'turn' in Ruby version
```

```
    {game, bot} = turn(bot.kind, bot, game)
    mainloop(bot, game)
  end

  def awaken(bot, game) do
    spawn Bot, :mainloop, [bot, game]
  end

end
```

On the Ruby version, I didn't actually do any automated tests. But as this is an Elixir book, not a Ruby book, let's rememdy that for the Elixir version.

The simplest things to test are the grid functionality and the behavior of the bots. This dooesn't cover all possible bugs, but it is a significant start.

For the grid, we basically ask: Can we store a bot inside this two-dimensional structure? And can we retrieve it later?

```
defmodule GridTest do
  use ExUnit.Case

  test "put and get" do
    grid = %{}
    where = {:red, 3, 5}
    val = "foo"
    grid = Grid.put(grid, where, val)
    assert Grid.get(grid, where) == val
  end

  test "show cell" do
    grid = Grid.put(%{}, {:red, 1, 1}, "piece")
    assert Grid.show_cell(grid, 1, 2) == :ok
  end



end
```

For the bots, we can "seed" the grid with a small number of known pieces. Then we can test the ability of each bot to see its surroundings, to detect a flag, to move, to attack, and so on.

```
defmodule BotTest do
  use ExUnit.Case

  test "where" do
    bob = Bot.scout(:red, 1, 5)
    assert Bot.where(bob) == {1, 5}
  end

  test "enemy? true when two bots different colors" do
    bob = Bot.scout(:red, 1, 5)
    nik = Bot.scout(:blue, 2, 6)
    assert Bot.enemy?(bob, nik)
  end

  test "enemy? false when two bots same color" do
    bob = Bot.scout(:red, 1, 5)
    nik = Bot.scout(:red, 2, 6)
    refute Bot.enemy?(bob, nik)
  end


end
```

It's possible, of course, to perform even more tests. For example, we could capture the textual representation of the board as a string (rather than outputting it directly) so that we could test its value. In this exercise, I haven't chosen to do such elaborate testing.