The DRY Principle

Relevant Links

- Wikipedia page on DRY[1]
- DevIQ link for DRY[2]

Don't Repeat Yourself

DRY stands for "Don't Repeat Yourself". It is a principle that duplication should be avoided at all levels. This principle takes many guises:

- Creating automated tests prevent us from duplicating our testing efforts, manually recreating testing situations whenever a code changes.
- Code duplication should be avoided by introducing abstraction. This may mean creating a function that achieves the repeated tasks, creating an array of values and performing a loop over those values, or even introducing a whole new class.
- The duplication of "magic numbers" throughout the code can be avoided by the use of constants.
- Duplication of solutions to common problems when designing the program can be avoided by the use of design patterns, which are standardized solutions to these common problems.

Case study: Tic-tac-toe

We will consider these topics in the context of a specific example, namely the tic-tac-toe board that was part of earlier labs. Let us recall the individual components of that assignment:

- A board is represented as a flat array of the correct size to fit one entry for each board square.
- An `initializeBoard` method initializes the board by setting all squares to `null`.
- A `get` method returns the value at a board location provided via `row` and `column` indices (rather than a flat index).
- A `set` method sets the value at a board loation if it is not already set. It throws an error if the value is previously set.
- A `winner` method looks through the possible winning configurations to see if any of them suggests a winner, and if so it returns both the winner and the configuration.

[1] https://en.wikipedia.org/wiki/Don't_repeat_yourself
[2] http://deviq.com/don-t-repeat-yourself/

Initial Design Let's look at a basic implementation, and then we will iterate over its design.

```
function makeBoard() {
   var board;

   function initBoard() {
      board = [null, null, null, null, null, null, null, null, null];
   }

   function get(row, col) {
      if ( row === 0 && col === 0 ) {
         return board[0];
      }
      else if ( row === 0 && col === 1 ) {
         return board[1];
      }
      else if ( row === 0 && col === 2 ) {
         return board[2];
      }
      else if ( row === 1 && col === 0 ) {
         return board[3];
      }
      else if ( row === 1 && col === 1 ) {
         return board[4];
      }
      else if ( row === 1 && col === 2 ) {
         return board[5];
      }
      else if ( row === 2 && col === 0 ) {
         return board[6];
      }
      else if ( row === 2 && col === 1 ) {
         return board[7];
      }
      else {
         return board[8];
      }
   }

   function set(row, col, value) {
      if ( row === 0 && col === 0 ) {
         board[0] = value;
      } else if ( row === 0 && col === 1 ) {
         if (board[1] === null) {
            board[1] = value;
         } else {
            throw "error";
         }
      } else if ( row === 0 && col === 2 ) {
         if (board[2] === null) {
            board[2] = value;
         } else {
            throw "error";
         }
      } else if ( row === 1 && col === 0 ) {
         if (board[3] === null) {
```

```
            board[3] = value;
         } else {
            throw "error";
         }
      } else if ( row === 1 && col === 1 ) {
         if (board[4] === null) {
            board[4] = value;
         } else {
            throw "error";
         }
      } else if ( row === 1 && col === 2 ) {
         if (board[5] === null) {
            board[5] = value;
         } else {
            throw "error";
         }
      } else if ( row === 2 && col === 0 ) {
         if (board[6] === null) {
            board[6] = value;
         } else {
            throw "error";
         }
      } else if ( row === 2 && col === 1 ) {
         if (board[7] === null) {
            board[7] = value;
         } else {
            throw "error";
         }
      } else {
         if (board[8] === null) {
            board[8] = value;
         } else {
            throw "error";
         }
      }
   }

   function winner() {
      var winner, player, values;

      if (checkLine(board[0], board[1], board[2]) != null) {
         player = board[0];
         values = [0, 1, 2];
         winner = { "player": player, "entries": values} ;
         return winner;
      } else if ( checkLine(board[3], board[4], board[5]) != null) {
         player = board[3];
         values = [3, 4, 5];
         winner = { "player": player, "entries": values };
         return winner;
      } else if (checkLine(board[6], board[7], board[8]) != null) {
         player = board[6];
         values = [6, 7, 8];
         winner = { "player": player, "entries": values };
         return winner;
      } else if (checkLine(board[0], board[3], board[6]) != null) {
```

```
            player = board[0];
            values = [0, 3, 6];
            winner = { "player": player, "entries": values };
            return winner;
        } else if (checkLine(board[1], board[4], board[7]) != null) {
            player = board[1];
            values = [1, 4, 7];
            winner = { "player": player, "entries": values };
            return winner;
        } else if (checkLine(board[2], board[5], board[8]) != null) {
            player = board[2];
            values = [2, 5, 8];
            winner = { "player": player, "entries": values };
            return winner;
        } else if (checkLine(board[0], board[4], board[8]) != null) {
            player = board[0];
            values = [0, 4, 8];
            winner = { "player": player, "entries": values };
            return winner;
        } else if (checkLine(board[2], board[4], board[6]) != null) {
            player = board[2];
            values = [2, 4, 6];
            winner = { "player": player, "entries": values };
            return winner;
        } else {
            return null;
        }
    }

    // We initialize the board, before returning.
    initBoard();

    return {
        reset: initBoard,
        set: set,
        get: get,
        winner: winner
    };
}
```

This solution has a number of problems.

1. Both `get` and `set` use the same logic to determine which array index corresponds to which combination of row and column. Having to keep these two pieces of logic in sync is a considerable addition to the reader's cognitive load.
2. To determine the correct index, a long chain of `if-then-else` statements is used. This is terribly repetitive, and also not easily extensible. What happens if the board size changes? Also testing the correctness of this code is considerably harder.
3. The `board` initializer hard-codes the length of the array, as well as the value used to represent an empty square (`null`). If something changes in the specifications, this would require changes in multiple places (for instance all the `null` values instead of just one).

4. The winner logic is very repetitive with a long chain of if–then–else statements. If we must change the format of the return value for example, we must now do so in multiple places. Or if the rules of the game change to allow different configurations as the winning configurations, this would also require considerable changes to the code.

   If you have multiple if clauses all performing similar steps, there is something wrong with the structure of your program.

Improvements Let us now consider some improvements:

1. The logic for determining the index corresponding to a row-column pair should become its own function. (This could not be technically done in the constraints of the assignment).
2. The dimensions of the board should be flexible, rather than hard-coded as 3. We could use local variables, but will instead add them as parameters to the initial function call, with defaults set to 3. The initBoard function should be adjusted to fill an array of a size determined by those dimensions. 3.The logic for determining the index, which is now its own function, should be able to account for the different possible dimensions. This requires some thinking to find a nice formula.
3. The determination of which triples could constitute a winning combination (or if in fact they would be triples at all) should be separate from the process that checks each particular triple to see if it does form a winning combination. We will create a separate process for that (and for now hard-code it for the 3x3 game; the reader might think about ways to make that function more flexible). This is an important consideration however: The code generating the cases is *decoupled* from the code that checks the cases.

Here's how the code may look like with these improvements.

```
function makeBoard(nrow, ncol) {
   var board, size, emptyVal;

   emptyVal = null;

   if (nrow == undefined) { nrow = 3; }
   if (ncol == undefined) { ncol = 3; }

   function initBoard() {
      var i;

      board = [];
      for (i = 0; i < nrow * ncol; i += 1) {
         board[i] = emptyVal;
      }
   }

   // returns the index at which the specific row-col combination corresponds
   function getIndex(row, col) {
```

```
    if (row < 0 || row >= nrow ||
        col < 0 || col >= ncol) {
      throw new Error('Dimensions out of bounds: (' + row + ', ' + col + ')');
    }

    return row * ncol + col; // Each row adds ncol entries to the array
}

function get(row, col) {
    return board[getIndex(row, col)];
}

function set(row, col, value) {
    var index;

    index = getIndex(row, col);
    if (board[index] !== emptyVal) {
      throw new Error('Value already set at: ' + ('' + row + ', ' + col + ')');
    }

    board[index] = value;

    return this;
}

function getConfigs() {
    return [
      [0, 1, 2], [3, 4, 5], [6, 7, 8],     // rows
      [0, 3, 6], [1, 4, 7], [2, 5, 8],     // columns
      [0, 4, 8], [2, 4, 6]                 // diagonals
    ];
}

function winner() {
    var winner, player;
    var allConfigs, config, i;
    var values, j;

    allConfigs = getConfigs();

    for (i = 0; i < allConfigs.length; i += 1) {
      config = allConfigs[i];

      values = [];
      for (j = 0; j < config.length; j += 1) {
        values[j] = board[config[j]];
      }
      // Will learn how to not hard-code this later.
      // Or change checkLine to take an array of values.
      player = checkLine(values[0], values[1], values[2]);
      if (player !== null) {
        return { player: player, entries: config };
      }
    }

    // No winner
```

```
        return null;
    }

    // We initialize the board, before returning.
    initBoard();

    return {
        reset: initBoard,
        set: set,
        get: get,
        winner: winner
    };
}
```