CamelCoders Test Plan

Aidan McNay (acm289) • Andro Janashia (aj454) • Henry Toll (hht26)

When testing CheckCamelMate, we utilized a combination of black-box and glass-box testing. We first took a black-box approach, testing all of the producer-consumer combinations, as well as all of the boundary cases that we determined. Once we had achieved all of the test cases we believed, we then went back and utilized coverage reports to perform glass-box testing, ensuring to get as much coverage as possible (100%, with the exception of GUI-related functions). This achieved a test suite utilizing both methodologies to give us a high degree of confidence in our program. We were unable to utilize randomized testing in this project, due to the difficulty of determining whether a random move would be valid or not outside of our already-implemented logic under test.

Our test suite was entirely implemented using OUnit, with the exception of GUI-related functions, which were instead manually verified through interactions in the GUI. This allowed as much of our project as possible to be tested automatically, allowing for ease of testing from just running `dune test`.

Utilities

Our game relies on two underlying data types; a `Utils.Move.t` (variant of `Up`, `Down`, `Left`, and `Right`) representing a move on the chess board, as well as a `Utils.Location.t` (containing a `column` and a `row`), representing a location on the board.

While moves are just a data abstraction, locations have functions that can modify them, including creating a location from a row and column (a "producer"), accessing those members from a location ("consumer"), and creating new locations from old locations by applying moves (both a "producer" and "consumer"). We made sure to test these different combinations, as well as the case that a location was off the board (raising an exception if attempted). Once we had tested these combinations, we utilized glass-box testing to achieve full coverage; our only remaining coverage needed was the different cases that could identify whether a location was off the board. Once we obtained full coverage for these utilities, we had satisfied both our black-box and glass-box testing methodologies, giving us confidence that our implementation was correct. This unit testing was done before proceeding with the larger system, so that if any bugs arose later, we could narrow the scope to new code, confident that our utilities were correct.

The only function that wasn't tested for our utilities was **`color_of_loc`**; this evaluated to the color that should be used on a chessboard for a provided location. The color returned was a data abstraction as part of the Bogue library, and was therefore left untested by OUnit (although manually verified with visual inspection in our GUI).

Pieces

Once our underlying utilities were tested, we moved on to testing the **`Piece`** library, specifically the **`Piece.Pieces.t`** type, which represented an instance of a chess piece (containing information about its color, type (**`Pawn`**, **`Knight`**, etc.), location, and what its valid moves were).

Similar to with locations, we had an initialization function to create a piece (a "producer"), many functions to get attributes of a piece, such as its type and color ("consumers"), and the ability to return a new piece with attributes changes, such as location if we're moving a piece (both a "producer" and "consumer"). Our black-box testing checked that all of these producers and consumers interacted as we'd expect; for example, if we created a piece as **`Black`**, then **`get_color`** with that piece would evaluate to **`Black`**. Our testing in this section additionally focused around the function **`get_valid_moves`**, which evaluated to all the valid moves a piece could take given the other surrounding pieces; we wanted to make sure that each piece had the correct movement logic before going up to higher levels of abstraction for the entire chess board. This was done using a functor that took in the moves we'd expect for the given scenario, and created the test for us, allowing for high code reusability.

From here, we additionally engaged in glass-box testing, checking to make sure that we had full coverage of testable functions. The only coverage we needed to make sure to add was regarding all the possible moves a piece could take (including when a same-color piece was in the way, a different-color piece was in the way, all the possible moves that a knight could take, and double-moves for pawns on their first move), as well as cases where a piece's ability to make a move would change (such as pawns not being able to move 2 if they weren't on the starting rank). Once we obtained full coverage for these utilities, we had satisfied both our black-box and glass-box testing methodologies, giving us confidence that our implementation was correct. Similar to before, this unit testing was done before proceeding to the overall system, as to limit the scope of code needing to be debugged if any further bugs arose.

The only function that wasn't tested for pieces was **`to_image`**, which evaluated to a **`Bogue.Widget.t`** representation of the image of a piece (utilizing icons obtained from Wikimedia Commons). Since this is a largely graphical function, it could not be tested automatically with OUnit; however, we did some manual ad-hoc testing with small **`Bogue`** windows to make sure that the images were loaded properly, as well as visually verifying once the entire GUI for the chessboard was implemented.

Chessboard

Lastly, we implemented the functionality for a **`Board.Chessboard.t`**, which represents an entire chessboard, including the pieces on it and moves that have been made. This includes a few "producers" (the initial state **`initial`**, as well as the ability to make a new board with the **`mk_board`** function), some "consumers" that get information about the board (such as **`in_checkmate`** to see if a color is in checkmate, as well as **`last_move`** to get the last move made on the board as a **`Board.Move_record.t`**, which is another data abstraction with one producer and multiple consumers that was tested as part of testing **`Board.Chessboard.t`**). However, the main function used is the **`make_move`** function, which takes in a current chessboard, and updates to a new chessboard with the move made, evaluating to **`Board.Chessboard.Invalid_move`** is the move isn't valid, or **`Board.Chessboard.Puts_in_check`** if the move would put the moves in check.

To test these, similar to before, we first started with black-box testing, testing combinations of these getters and setters. This was done using a functor that took in the current board state, and tried to execute the specified move, verifying it against the expected output (either a **`Board.Move_record.t`**, or the raising of **`Board.Chessboard.Invalid_move`** or **`Board.Chessboard.Puts_in_check`**, with these three outcomes implemented as the **`outcome`** variant). Beyond simple moves (done using the **`initial`** board from before for code reuse), we also made sure to test corner cases, such as checking/checkmating, promotion, en passant, and castling in their own separate testing file, as well as the corner cases for the validity of these moves (such as that a castle isn't valid when the king is in check, or when the king or rook has already moved)

From here, we proceeded to use glass-box testing to achieve full coverage or testable functions. While much of our code had already been tested, we noticed that we had yet to achieve full coverage in our **`Board.Alg_notation`** module, responsible for

generating the algebraic notation for moves. Here, we implemented more test cases that tested the notation made when the piece being moved was ambiguous (when two pieces could move to the same square, meaning normal notation wasn't sufficient). After this, similar to before, we had achieved full coverage while testing all of the corner cases we could identify, giving us confidence that our functionality was correct.

Like before, we left a few functions not completely tested. The first was **`image_at_loc`**, which got the **`Bogue.Widget.t`** representing the image at a given location on the board; like with other GUI functions, we couldn't test this widget in OUnit2, but rather visually identified in our final GUI that the images at given locations represented the pieces at those location. In addition, our code utilized the **`Prompt`** library, used for prompting the user for what piece they want to promote to when necessary. Since this involves user interaction, it couldn't be automatically tested by OUnit2; however, we similarly verified that a pawn promoted to our response to the prompt in real games. Initially, the **`Prompt`** library always prompted the user; however, for testing purposes, we modified it so that it would default to a **`Piece.Types.Queen`** when not running a real game, so that we could still test the rest of our promotion logic.

GUI

The last part of our system was the GUI, where players click to move pieces on a chessboard. These components proved difficult to test with OUnit; they all involve abstractions implemented by the Bogue library, making it difficult to have enough visibility to test them, as well as having a strong visual dependence. Because of this, these functions were not tested as part of our test suite, but rather inspected and debugged visually when running our program. Our GUI is implemented using our game logic as the primary way for users to interact; since our game logic didn't depend on this code, we still felt confident in the functionality of our underlying game logic from the rest of our tests.