Robo-Wars Testing Document

Team D4

Nickolas Gough, Scott Hebert, Janelle Hindman, Yige Huang, and Tushita Patel

Project: Robo-Wars

Dr. Christopher Dutchyn

CMPT 370 Term I 2016/2017

Executive Summary

As part of test-driven design (TDD) protocols, a testing document covering the testing interfaces and procedures should be created before the code construction step. The overall approach for the testing interfaces will be creating unit tests in each Class using Main methods—this is known as static public void Main (SPVM) testing.

Given that there is a lack of infrastructure to perform rigorous user interface testing, the primary testing interfaces for the game will be focused in the Interpreter and Board components. Unlike the human player, the AI scripts are both well-structured and must be parsed by the computer semantically. The Board component is the central hub of communication in the system architecture, and therefore it requires more thorough testing than other components. The other components are either highly human-centric (e.g., the Flow component), or primarily data structures (e.g., the Pieces component), and so their Classes are not expected to require as much unit testing, and may require thorough testing by a human tester after code construction.

Finally, working through the testing apparatus for the game revealed that the optional requirement for asynchronous network will not be realized. While the networking may have been able to be created, there is not sufficient time to test the robustness of any networking elements.

Contents

1.0 Introduction

With a complete design, the focus of the project turns to ensuring that the design will perform as expected. The prevalent contemporary testing paradigm to gain confidence in system robustness is test-driven design (TDD). This requires that testing interfaces—and, if possible, testing procedures—be created before the code construction step. The overall testing pattern will be discussed and explained, and then details of the primary testing interfaces will be detailed.

1.1 Primary Test Pattern: SPVM Unit Testing

In terms of testing patterns, a common and fairly simple testing pattern that will be extensively used is the so-called static public void Main (SPVM) approach of unit testing. In this pattern, each Class is given the ability to be called in isolation of the entire system (the "Main" method), and if it is called in this manner, it runs test scripts designed to ensure that the system works as expected.

There are a few reasons why SPVM testing was chosen as the primary test pattern. The first is that it is very straightforward to do. SPVM is familiar and is not expected to create complications and difficulties. Another reason to concentrate on SPVM testing is that in an independent components (IC) architecture, interdependencies among the various components are minimized. As noted in the design document, this allows for unit testing as the primary testing pattern.

2.0 Testing Interfaces

Each component of the system will need to be tested in some way, but not all of them are well-suited to unit testing. This section is primarily concerned with unit testing of each significant component. Justification will be given for each component that requires testing. Where unit testing is not possible, this section briefly outlines strategies for user testing of significant components, which by necessity must come after the code construction and system integration has begun.

For each component and its important classes, this section does not detail tests for accessor and mutator methods for those classes ("getter" and "setter" methods). Each method that requires getting or setting a field has an implied set of basic Test cases, which are not described here for brevity's sake.

2.1 Interpreter Component Testing Interface

2.1.1 Interpreter Component Description

As described in the design document (see **Section 2.5** of the design document), the Interpreter is a component of the software system that handles the parsing of the computer-controlled player programs, as found in the Robot Archive. It contains the Interpreter Class, and helper classes in the form of the AI Class and the Mailbox Class. It is responsible for initializing the AI at the beginning of a match, and running each AI's program every turn. It will need to transmit the results of its parsing to the Board component when necessary, and handle errors within the user-created programs gracefully.

2.1.2 Significance of the Interpreter

The Interpreter comprises a significant portion of the game system, in that it is what enables AI teams to exist during the match—a valuable feature of any computer game. Each AI runs according to a program that was created by a user. As with any piece of software, user input must be treated with suspicion.

Without a robust way of handling these AI programs, there is a great risk that malformed, mistyped, or malicious programs could cause the software to lock up, crash, or otherwise disrupt the play experience for the human user. This makes testing the Interpreter paramount to the success of the system.

2.1.3 Interpreter Class Testing Plan

Unit tests will provide the backbone of the testing plan for the Interpreter Class. Each AI program is written in a Forth-like language that performs its operations using a stack data structure. These operations are encoded into 38 base "words," which provide the base functionality for the Interpreter, known as the Word Interface. Thus, each method in the Word Interface that interacts with the stack must be carefully tested to ensure that the desired values are pushed to and popped from the stack in the correct order.

Since the operations are stack-based, if one operation fails, it is expected that all subsequent operations have a high chance of failure as well. Rather than take the risk that the first failure could have a domino effect on the program, potentially causing an infinite loop, the Interpreter will immediately terminate an AI program turn if a vital (non-debug) operation fails, and will log the failure to the console as well as writing an error file to the filesystem. This will minimize the chance that a malformed AI will negatively affect the user: a software lock-up is more damaging than an AI that makes no move on its turn.

Any method in the Word Interface that is required to pop a value off the stack, unless otherwise noted, will require the following Test case:

> **Test case: The stack contains zero values.**
> Expected result: The stack will have no values, and an error will be logged.

This Test case is so common it is identified above for brevity. The other, individual Test cases for each method are below.

- `add()`:
    - **Test case: The stack contains fewer than two values.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values where one or both are not integers.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two integers on top.**
      Expected result: The top of the stack contains the correct result of the addition.
- `subtract()`:
    - **Test case: The stack contains fewer than two values.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values where one or both are not integers.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two integers on top.**
      Expected result: The top of the stack contains the correct result of the subtraction.

- `multiply()`:
    - **Test case: The stack contains fewer than two values.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values where one or both are not integers.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two integers on top.**
      Expected result: The top of the stack contains the correct result of the multiplication.

- `divideRemain()`:
    - **Test case: The stack contains fewer than two values.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values where one or both are not integers.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two integers on top that divide evenly.**
      Expected result: The top of the stack contains the correct result of the division, and the second value from the top is `0`.
    - **Test case: The stack contains two integers on top that do not divide evenly.**
      Expected result: The top of the stack contains the correct result of the division, and the second value from the top is the remainder.

- `and()`:
    - **Test case: The stack contains fewer than two values.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values where one or both are not boolean values.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two boolean values on top.**
      Expected result: The top of the stack contains the correct result of the `and()` operation as a boolean, `true` or `false`.

- `or()`:
    - **Test case: The stack contains fewer than two values.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values where one or both are not boolean values.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two boolean values on top.**
      Expected result: The top of the stack contains the correct result of the `or()` operation as a boolean, `true` or `false`.

- `invert()`:
    - **Test case: The stack contains one non-boolean value on top.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains one boolean value on top.**
      Expected result: The top of the stack contains the opposite boolean value.
- `duplicate()`:
    - **Test case: The stack contains one value of any type on top.**
      Expected result: The top of the stack contains two of the original value.
- `drop()`:
    - **Test case: The stack contains one value of any type on top.**
      Expected result: The value has been removed from the stack, and nothing has been pushed to the stack.
- `swap()`:
    - **Test case: The stack contains one value of any type on top.**
      Expected result: The value has been replaced at the top of the stack.
    - **Test case: The stack contains two values of any type on top.**
      Expected result: The second value is on the top of the stack, and the first is just below it.
- `rotate()`:
    - **Test case: The stack contains one value of any type on top.**
      Expected result: the value has been replaced at the top of the stack
    - **Test case: The stack contains two values of any type on top.**
      Expected result: The two values have been replaced at the top of the stack in their original order.
    - **Test case: The stack contains three values of any type on top.**
      Expected result: The three values will be replaced on the stack, with the second at the bottom, the first in the middle, and the third at the top.
- `greaterThan()`:
    - **Test case: The stack contains fewer than two values.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values on top where both are not the same type.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values on top where both are the same type but not comparable.**
      Expected result: The stack will be cleared, and an error will be logged.

- **Test case: The stack contains two values on the stack of the same type and are comparable**
  Expected result: The top of the stack contains the correct result of the `greaterThan()` operation as a boolean, `true` or `false`.

- `greaterThanEqual()`:

  - **Test case: The stack contains fewer than two values.**
    Expected result: The stack will be cleared, and an error will be logged.
  - **Test case: The stack contains two values on top where both are not the same type.**
    Expected result: The stack will be cleared, and an error will be logged.
  - **Test case: The stack contains two values on top where both are the same type but not comparable.**
    Expected result: The stack will be cleared, and an error will be logged.
  - **Test case: The stack contains two values on the stack of the same type and are comparable**
    Expected result: The top of the stack contains the correct result of the `greaterThanEqual()` operation as a boolean, `true` or `false`.

- `lessThan()`:

  - **Test case: The stack contains fewer than two values.**
    Expected result: The stack will be cleared, and an error will be logged.
  - **Test case: The stack contains two values on top where both are not the same type.**
    Expected result: The stack will be cleared, and an error will be logged.
  - **Test case: The stack contains two values on top where both are the same type but not comparable.**
    Expected result: The stack will be cleared, and an error will be logged.
  - **Test case: The stack contains two values on the stack of the same type and are comparable**
    Expected result: The top of the stack contains the correct result of the `lessThan()` operation as a boolean, `true` or `false`.

- `lessThanEqual()`:

  - **Test case: The stack contains fewer than two values.**
    Expected result: The stack will be cleared, and an error will be logged.
  - **Test case: The stack contains two values on top where both are not the same type.**
    Expected result: The stack will be cleared, and an error will be logged.
  - **Test case: The stack contains two values on top where both are the same type but not comparable.**
    Expected result: The stack will be cleared, and an error will be logged.
  - **Test case: The stack contains two values on the stack of the same type and are comparable.**

Expected result: The top of the stack contains the correct result of the `lessThanEqual()` operation as a boolean, `true` or `false`.

- `equal()`:
    - **Test case: The stack contains fewer than two values.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values on top where both are not the same type.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values on top where both are the same type but not comparable.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values on the stack of the same type and are comparable.**
      Expected result: The top of the stack contains the correct result of the `equal()` operation as a boolean, `true` or `false`.

- `notEqual()`:
    - **Test case: The stack contains fewer than two values.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values on top where both are not the same type.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values on top where both are the same type but not comparable.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: The stack contains two values on the stack of the same type and are comparable.**
      Expected result: The top of the stack contains the correct result of the `notEqual()` operation as a boolean, `true` or `false`.

- `if()`:
    - **Test case: The `test` block in the program is empty.**
      Expected result: The `if` and `else` blocks are ignored, and the `then` block is executed.
    - **Test case: The stack does not contain values to execute the `test` block properly.**
      Expected result: The `if` and `else` blocks are ignored, and the `then` block is executed.
    - **Test case: The stack contains values to execute the `test` block properly, and the value returned is not a boolean.**
      Expected result: The `if` and `else` blocks are ignored, and the `then` block is executed.
    - **Test case: The stack contains values to execute the `test` block properly, and the result returned is a `true` value boolean.**
      Expected result: The code in the `if` block is executed, followed by the code in the `then` block.

- **Test case: The stack contains values to execute the `test` block properly, and the result returned is a `false` value boolean.**
  Expected result: The code in the `else` block is executed, followed by the code in the `then` block.

- `while()`:

  - **Test case: The `finished` block of the program is empty.**
    Expected result: The `body` is ignored, and the `until` block is executed.
  - **Test case: The stack does not contain values to execute the `finished` block properly.**
    Expected result: The `body` is ignored, and the `until` block is executed.
  - **Test case: The stack contains values to execute the `finished` block properly, and the value returned is not a boolean.**
    Expected result: The `body` is ignored, and the `until` block is executed.
  - **Test case: The stack contains values to execute the `finished` block properly, and the value returned is a `true` value boolean.**
    Expected result: The code in the `body` is ignored, and the `until` block is executed.
  - **Test case: The stack contains values to execute the `finished` block properly, and the value returned is a `false` value boolean, and the `finished` block will never return `true`.**
    Expected result: The code in the `body` is executed repeatedly, `until` the Interpreter detects that the time limit has been exceeded, and then the Interpreter ends the piece's turn.
  - **Test case: The stack contains values to execute the `finished` block properly, and the value returned is a `false` value boolean, and the `finished` block will eventually return `true`.**
    Expected result: The code in the `body` is executed as many times as the `finished` block evaluates to `false`, and then the `until` block is executed.

- `for()`:

  - **Test case: The `end` or `start` expressions of the program are missing.**
    Expected result: The `body` will only be executed once.
  - **Test case: The value of the `end` expression is less than or equal to the `start` expression.**
    Expected result: The `body` will only be executed once.
  - **Test case: the value of the `start` expression is less than or equal to the `end` expression**
    Expected result: The `body` will be executed a number of times, incrementing the iterator, until the `start` expression is greater than the `end` expression, and the iterator will be destroyed.
  - **Test case: The value of the `start` expression is less than or equal to the `end` expression, and the `body` code of the program contains the leave statement.**

Expected result: The `body` will be executed a number of times, incrementing the iterator, until the `start` expression is greater than the `end` expression or the leave statement in the `body` code is reached, and the iterator will be destroyed.

- `declareVar()`:
    - **Test case: `declareVar()` is passed the name of a variable that does not currently exist.**
      Expected result: A new UserVariable has been created in the Interpreter with the given name.
    - **Test case: `declareVar()` is passed the name of a variable that currently exists.**
      Expected result: No new UserVariables are added to the Interpreter.

- `declareWord()`:
    - **Test case: `declareWord()` is passed the name of a word that does not currently exist.**
      Expected result: A new UserWord has been created in the Interpreter with the given name and replacement values (even if the replacement values are empty).
    - **Test case: `declareWord()` is passed the name of a word that currently exists.**
      Expected result: The existing UserWord is updated to use the given replacement values (even if the replacement values are empty).

- `random()` :
    - **Test case: There is one value on the stack which is not an integer.**
      Expected result: The stack will be cleared, and an error will be logged.
    - **Test case: There is one value on the stack which is an integer.**
      Expected result: A random integer between `0` and the popped value (exclusive) has been pushed to the stack.

- `dotPrint()`:
    - **Test case: There is one value on the stack.**
      Expected result: A string representation of the value has been printed to the console.

The `qHealth()`, `qHealthLeft()`, `qMoves()`, `qMovesLeft()`, `qAttack()`, `qRange()`, `qTeam()` and `qType()` methods are essentially accessor methods that must interact with the Board component. Their tests must verify that the value retrieved from the Board and pushed to the stack is correct. If for some reason the piece cannot be found on the board, nothing will be pushed to the stack.

The `turn()`, `move()`, `shoot()`, `check()`, `scan()` and `identify()` methods all require interaction with the Board. They will query the board and their correctness depends heavily on the Board's implementation of its methods. If the Board or Piece cannot be accessed, these methods should

exit with an error code. The methods that interact with the stack will need to verify that the values on the stack are of the correct number and type, and that if any value is pushed, the correct value is pushed.

Outside of the 38 Word Interface methods providing standard operations, the Interpreter has one method that must be tested extensively: the `parse()` method. This method takes in a single string term and must parse it into the correct value type, which may be a string, integer or boolean data type, or some internally important value, such as the name of a user-defined variable or the name of a Word Interface method to call. `parse()` will need some transparent-box testing, in order to make sure that it is performing operations in the correct order as it parses terms. Depending on the context in which `parse()` is called, it may need to call certain methods after it parses a term. It would require the following Test cases:

- `parse(String line)`:
    - **Test case: `parse()` is passed an empty string.**
      Expected result: `parse()` will perform no action.
    - **Test case: `parse()` is passed a string which evaluates to an integer.**
      Expected result: `parse()` will correctly convert the string to an integer, and perform the appropriate operation with it.
    - **Test case: `parse()` is passed a string which evaluates to a boolean value `true` or `false`.**
      Expected result: `parse()` will correctly convert the string to a boolean value, and perform the appropriate operation with it.
    - **Test case: `parse()` is passed a string which begins with a "`.`" character.**
      Expected result: `parse()` will correctly convert the string into a cleaned string, without the leading character, and perform the appropriate operation with it.
    - **Test case: `parse()` is passed a string which evaluates to the name of an existing UserWord.**
      Expected result: `parse()` will retrieve the correct UserWord and perform the appropriate operation with it.
    - **Test case: `parse()` is passed a string which evaluates to the name of an existing UserVariable.**
      Expected result: `parse()` will retrieve the correct UserVariable and perform the appropriate operation with it.
    - **Test case: `parse()` is passed a string which evaluates to the name of an existing method in the Word Interface.**
      Expected result: `parse()` will call the correct method.

2.1.4 AI Class Testing Plan

The AI Class is primarily for holding the program data. When an instance is created, it takes in a raw program as a string, strips the comments out of it, and splits it into initialization and play sections. Most of its methods are just accessors for the initialization and play programs, but the AI class must be tested

to make sure that its raw program is correctly split when it is created. If a program contains no initialization or no play components, those fields should be empty. The most important method in this class to test is `stripComments()`, which will remove comments (denoted by "(" and ")" characters) from the programs before they are stored. Its Test cases are detailed below:

- `stripComments(String line)`:
    - **Test case: `stripComments()` is passed a string which contains no "(" and ")" characters.**
      Expected result: `stripComments()` returns the unmodified string.
    - **Test case: `stripComments()` is passed a string which contains a "(" character with no matching ")" character.**
      Expected result: `stripComments()` removes only the stray "(" character, and returns the modified string.
    - **Test case: `stripComments()` is passed a string which contains a ")" character with no matching "(" character.**
      Expected result: `stripComments()` removes only the stray ")" character, and returns the modified string.
    - **Test case: `stripComments()` is passed a string containing sets of matched "(" and ")" characters.**
      Expected result: `stripComments()` removes the "(" and ")" characters, as well as all the characters between them, and returns the modified string.

2.1.5 Mailbox Class Testing Plan

The Mailbox Class is used to hold messages that the various programs might pass between each other in order to communicate about the Board state. Thus, it must be tested carefully to ensure that messages are being added to and removed from the Mailbox correctly, and that it is correctly reporting its status. Its Test cases are detailed below:

- `hasMessage(String pieceID)`:
    - **Test case: The `messages` field of the Mailbox is empty.**
      Expected result: `hasMessage()` returns `false`.
    - **Test case: The `messages` field of the Mailbox is not empty, and `hasMessage()` is passed an empty string.**
      Expected result: `hasMessage()` returns `false`.
    - **Test case: The `messages` field of the Mailbox is not empty, and `hasMessage()` is passed a string which matches none of the `pieceID` s of the messages it contains.**
      Expected result: `hasMessage()` returns `false`.
    - **Test case: The `messages` field of the Mailbox is not empty, and `hasMessage()` is passed a string which matches with at least one of the `pieceID` s of the messages it contains.**
      Expected result: `hasMessage()` returns `true`.

- `receiveMessage(String pieceID)`:
    - **Test case: The `messages` field of the Mailbox is empty.**
      Expected result: `receiveMessage()` returns an empty string.
    - **Test case: The `messages` field of the Mailbox is not empty, and `receiveMessage()` is passed an empty string.**
      Expected result: `receiveMessage()` returns an empty string.
    - **Test case: The `messages` field of the Mailbox is not empty, and `receiveMessage()` is passed a string which matches none of the `pieceID` s of the messages it contains.**
      Expected result: `receiveMessage()` returns an empty string.
    - **Test case: The `messages` field of the Mailbox is not empty, and `receiveMessage()` is passed a string which matches with exactly one of the `pieceID` s of the messages it contains.**
      Expected result: `receiveMessage()` returns the message matching the `pieceID`, with the `pieceID` removed from the string, and removes the message from the Mailbox.
    - **Test case: The `messages` field of the Mailbox is not empty, and `receiveMessage()` is passed a string which matches with more than one of the `pieceID`s of the messages it contains.**
      Expected result: receiveMessage() returns the message matching the `pieceID` which was chronologically received first, with the `pieceID` removed from the string, and removes the message from the Mailbox.

- `sendMessage(String pieceID, String message)`:
    - **Test case: `sendMessage()` is passed an empty string as either of its parameters.**
      Expected result: `sendMessage()` delivers no messages and returns `false`.
    - **Test case: `sendMessage()` is passed an invalid `pieceID` string.**
      Expected result: `sendMessage()` delivers no messages and returns `false`.
    - **Test case: `sendMessage()` is passed the `pieceID` of a Mailbox that is full.**
      Expected result: `sendMessage()` delivers no messages and returns `false`.
    - **Test case: `sendMessage()` is passed the `pieceID` of a Mailbox that is not full.**
      Expected result: `sendMessage()` delivers the message using the matching Mailbox's `addMessage()` method, and returns `true`.

2.2 Board Component Testing Interface

2.2.1 Board Component Description

As described in the design document (see **Section 2.3** of the design document), the central component of the architecture for the actual play of the game is the Board component. It is responsible for maintaining the overall board state, directing the flow of the game, and communicating that state and flow to the other components. It is expected that most other components communicate with this component. As such, the Board component is responsible for tracking the teams and their pieces, ensuring that each play is valid according to the game rules, and that the change to the state of the

board is propagated to the Display and Logger components. The Board component is also responsible for representing the board and expressing the state of the board according to this representation. Looked at from an MVC architectural perspective, the Board has some of the functions of a Controller, but also some aspects of the Model. One of the primary inputs to the Board component comes from the Event Handler component. The board component notifies the Logger and the Display after each action so that each may update according to how the game board state has changed and which actions have been performed.

2.2.2 Significance of the Board

The Board class is significant because it directs the flow of the game, stores the overall state of the game board, and updates the Display and Logger. Also, the Board class is centralized in that it is the communication link between most of the other components for game updates within the system. It is important to assert that the Board class functions correctly to ensure the state of the game board remains stable, other components receive correct updates, the flow of the game is directed correctly and smoothly, and the game rules are adhered to.

The Board component will be tested to ensure that a piece is correctly moved, all pieces occupying a shot space are dealt the correct damage, the next active piece is correctly determined, scanning a space or area is performed correctly, spaces are correctly converted between their absolute and relative representations, and a space is correctly determined to be inside or outside of the board's boundaries. Extensive tests will also be conducted on the `reduce()` method of the board class, which takes a hexagon coordinate and reduces it to its minimal representation. Testing the `reduce()` method will include testing correct and incorrect hexagon coordinate inputs and testing sensitive hexagon coordinate inputs.

2.2.3 Board Class Testing Plan

The Board class is designed to model the game board. The Board class stores an array of the coordinates of the hexagon spaces at which the game pieces are located, an array of the coordinates of the hexagon spaces that are the boundary limits of the game board, an array of the game pieces, an array of the teams, the enumeration of the current game piece that is active, the logger, and the display. The Board class not only stores the necessary coordinates and pieces, but also manipulates the position of each piece according to direct instruction from the event handlers in the Flow component while also checking that each action is valid.

The Test cases for the methods in the Board Class are listed below.

- `Board()`:
    - **Test case: Initialize the Board.**
      Expected result: The correct number of teams and pieces are initialized and each team has the correct number and type of pieces. The Logger is initialized and present. The Display is initialized and present.
- `movePiece(PieceEnum piece, HexCoord coord)`:
    - **Test case: The coordinate represents a valid coordinate within bounds.**
      Expected result: The piece is moved to the coordinate.

- **Test case: The coordinate represents a valid coordinate not within bounds.**
  Expected result: The piece is not moved at all and the error is handled.
- **Test case: The coordinate does not represent a valid coordinate.**
  Expected result: The piece is not moved at all and the error is handled.
- **Test case: The piece parameter represents a valid piece.**
  Expected result: The correct piece is moved and no other piece is.
- **Test case: The piece parameter does not represent a valid piece.**
  Expected result: No piece is moved and the error is handled.
- **Test case: The travel distance is less than the piece's remaining mobility points.**
  Expected results: The piece is moved to the coordinate.
- **Test case: The travel distance is equal to the piece's remaining mobility points.**
  Expected results: The piece is moved to the coordinate.
- **Test case: The travel distance is greater than the piece's remaining mobility points.**
  Expected results: The piece is not moved and the error is handled.
- **Test case: A valid move is performed.**
  Expected results: The distance is subtracted from the piece's mobility points.
- **Test case: An invalid move is performed.**
  Expected results: The piece's mobility points remain unchanged.

- `shootSpace(PieceEnum piece, HexCoord coord)`:

  - **Test case: Shoot a space inside the range of the piece's range points.**
    Expected results: The shot is performed correctly.
  - **Test case: Shoot a space outside the range of the piece's range points.**
    Expected results: No pieces are damaged and the error is handled.
  - **Test case: Shoot an unoccupied space.**
    Expected results: No piece is dealt any damage.
  - **Test case: Shoot a space occupied by one piece.**
    Expected results: Only the piece occupying the space is dealt the correct damage.
  - **Test case: Shoot a space occupied by more than one piece.**
    Expected results: All pieces occupying the space are dealt the correct damage.
  - **Test case: The piece shoots are second time following a first shot.**
    Expected results: The shot is not performed and the error is handled.

- `nextPiece()`:

  - **Test case: Two rounds of game play are simulated for 2, 3, and 6 teams.**
    Expected results: The next active piece is correctly determined each turn.
  - **Test case: A piece no longer alive is next in line to be active.**
    Expected results: The piece is skipped and the next piece is correctly determined.
  - **Test case: The next team does not possess any live pieces.**
    Expected results: The next active team and piece is correctly determined.

- `scanArea(PieceEnum piece, HexCoord coord)`:

  - **Test case: The coordinate represents a valid coordinate within bounds.**
    Expected results: The correct scan is performed.
  - **Test case: The coordinate represents a valid coordinate outside bounds.**

Expected results: The scan is not performed and an exception is thrown.
    - **Test case: The coordinate represents an invalid coordinate.**
      Expected results: The scan is not performed and an exception is thrown.
    - **Test case: The scan is performed.**
      Expected results: The scan returns coordinates that are occupied.
    - **Test case: The scan is performed.**
      Expected results: The scan does not return coordinates that are not occupied.
    - **Test case: The scan is performed.**
      Expected results: The scan does not return coordinates outside the piece's range.

- `scanSpace(HexCoord coord)`:

    - **Test case: The coordinate represents a valid coordinate within bounds.**
      Expected results: The correct scan is performed.
    - **Test case: The coordinate represents a valid coordinate outside bounds.**
      Expected results: The scan is not performed and an exception is thrown.
    - **Test case: The coordinate represents an invalid coordinate.**
      Expected results: The scan is not performed and an exception is thrown.
    - **Test case: The space is unoccupied.**
      Expected results: The scan returns an empty list.
    - **Test case: The space is occupied by one piece.**
      Expected results: The scan returns a list of only the piece occupying the space.
    - **Test case: The space is occupied by more than one piece.**
      Expected results: The scan returns a list of all the pieces occupying the space.

- `absoluteToRelative(HexCoord coord)` and `relativeToAbsolute(HexCoord coord)`:

    - **Test case: The coordinate represents a valid coordinate within bounds.**
      Expected results: The correct conversion is performed.
    - **Test case: The coordinate represents a valid coordinate outside bounds.**
      Expected results: The conversion is not performed and an exception is thrown.
    - **Test case: The coordinate represents an invalid coordinate outside bounds.**
      Expected results: The conversion is not performed and the an exception is thrown.
    - **Test case: Convert a relative coordinate to an absolute coordinate.**
      Expected result: The correct absolute coordinate is returned.
    - **Test case: Convert an absolute coordinate to a relative coordinate.**
      Expected result: The correct relative coordinate is returned.
    - **Test case: Convert a relative coordinate to an absolute coordinate and back to a relative coordinate.**
      Expected result: The original relative coordinate is returned.
    - **Test case: Convert an absolute coordinate to a relative coordinate and back to an absolute coordinate.**
      Expected result: The original absolute coordinate is returned.

- `isInBounds(HexCoord coord)`:

- **Test case: The coordinate represents a valid coordinate within bounds.**
  Expected result: `isInBounds()` returns `true`.
- **Test case: The coordinate represents a valid coordinate outside bounds.**
  Expected result: `isInBounds()` returns `false`.
- **Test case: The coordinate represents an invalid coordinate.**
  Expected result: `isInBounds()` returns `false`.

2.2.4 HexCoord Class Testing Plan

The HexCoord class is designed to model the hexagon space elements on the game board. The HexCoord class is used to refer to the actual hexagon spaces on the game board and for storing the coordinate positions of the game pieces. The HexCoord class will also be used throughout the system when it is necessary to refer to a hexagon space on the game board, such as within the display component to update the location of a visual image of a game piece.

The HexCoord class is significant because it represents the coordinates of the hexagon spaces on the game board, but more so because the reduce method of the HexCoord class is designed to reduce the x, y, and z coordinates down to their minimal representation for simplification, which must function correctly for the rest of the implementation to work.

The Test cases for the HexCoord class are listed below.

- `reduce(HexCoord coord)`:
  - **Test case: `reduce()` is called on a coordinate representing the center of the board.**
    Expected result: The center is correctly reduced.
  - **Test case: `reduce()` is called on coordinates just beyond the center.**
    Expected result: The coordinates are correctly reduced.
  - **Test case: `reduce()` is called on coordinates between the center and boundaries.**
    Expected result: The coordinates are correctly reduced.
  - **Test case: `reduce()` is called on coordinates just before the boundaries.**
    Expected result: The coordinates are correctly reduced.
  - **Test case: `reduce()` is called on boundary coordinates.**
    Expected result: The coordinates are correctly reduced.
  - **Test case: `reduce(` is called on a coordinate outside the boundaries.**
    Expected result: The coordinate is not reduced and the error is handled.

2.3 Other Component Testing Interfaces

The Interpreter and Board components are the two most significant elements in the software system that will require unit testing. Other components may require some unit testing to a degree, but there are some components, such as the Display and the Flow, that cannot be automatically validated through testing alone. Often these are components that require some type of human input or oversight. Nevertheless, they must be tested in some way. Where unit testing can be applied to these components, the Test cases have been listed below. Otherwise, a description of how these components will need to be tested with human users will follow.

2.3.1 Display Component Testing Interface

2.3.1.1 Display Component Description

The Display component (see **Section 2.1** of the design document) is responsible for the user interface (UI) elements of the game. It primarily takes information from the Board and displays that information to the user. The Display is primarily a passive component and does not send information out to other components.

2.3.1.2 Significance of the Display

The Display component is significant because it is responsible for visualizing the state and actions of the system for the user. It is important to ensure correct visualization of the system so the user can interact with and use the system. The Display is also the only visually appealing component within the system. If the Display is not visually appealing, and does not display the data and actions adequately, the system will not succeed. As such, it is imperative to determine that each Screen within the Display is correctly initialized, is visually appealing, and displays the data and actions in a way that is easily interpreted.

2.3.1.3 Display Class Testing Plan

The Display Class must be tested to determine that it correctly initializes the various screens, correctly switches between screens, and that each screen is accurately displayed. However, much of this testing cannot be done automatically or in isolation from the other components, since much of the Display's output is visual and requires input from other components. Thus, the Display class will require user testing after code construction, instead of automated unit testing.

The constructor method for the Display class will need to be tested to ensure that it is initializing all of the screens correctly. When the software launches, it must construct all of the screens the user will see, initializing each as an instance of the Screen class (see **Section 2.3.1.4**). These Screens will not be modified after the software has initialized, so their initialization is the place where visual errors may occur. The Display constructor will also need to set the initial parameters of the UI elements contained in each Screen, such as buttons and sliders, so that they can be interacted with in a proper manner by the Flow component. All of these features must be verified by a human tester.

One of the Display Class's crucial methods, `switchTo()`, will need to be tested in context with the rest of the components. It is responsible for switching which Screen is currently being displayed to the user, and will only be called after the user has performed some action, such as clicking a button. This method should be tested by a human in order to make sure that a valid Screen is always displayed when an appropriate action is taken.

2.3.1.4 Screen and GameScreen Classes Testing Plan

The Screen class has been designed to model a screen within the system. Each screen will have a specific layout, but the layout will be initialized by the Display Class. Each screen will be assigned an enumeration that will be stored and for which there is an accessor. Each screen will use the Swing library for its display components, will inherit from JPanel, and holds all of the display elements for that screen, including buttons, images, and other control widgets. Each screen must be tested by a human after code construction to ensure that all possible interactions are handled correctly.

The GameScreen class inherits from the Screen class and has been designed to model the game screen. It contains all elements that a Screen does, including its ScreenEnum value and the Panels it contains. A collection of images for the game pieces will be stored in an array that will be indexed using

enumerations where applicable. The x and y offsets of the board will be stored to allow the correct portion of the board to display when the Player pans the board. Also, the log screen and the context menu that will be shown to the Player to allow the Player to select their action are to be stored.

The GameScreen is significant because it displays the game's state and actions to the players. This Screen is the most important of all the Screens because this Screen is where the game is visualized. Every action within the game is visualized here. Much of the user's time and concern will be focused in this Screen, so this Screen must function correctly. If the game is not visualized correctly, the system will not succeed.

The GameScreen constructor must be tested on a match containing 2, 3 and 6 teams in order to make sure that the appropriate coloured images are loaded, that the board is correctly initialized and positioned, and that the UI elements are instantiated with correct parameters. These features cannot be automatically tested, and require a human tester to validate them.

The tester must then perform normal actions within the course of a map in order to test that the GameScreen's `update()` method is working properly. The `update()` method will redraw the screen according to important changes in the state of the board, such as a piece moving or shooting or the user panning the board. This will also allow the tester to verify that the GameScreen's `movePiece()` and `shootSpace()` methods are displaying changes correctly. These methods should not display invalid states of pieces, such as pieces which are out of bounds. The Board component should validate this information before passing it to the Display.

Other tests that must be performed with a human tester include testing to make sure that the GameScreen properly updates the screen when a game turn switches from one player to another. This requires that the display of the board should be centered on the active piece, that the piece is of the correct colour, and that the interface elements, such as the log window, are displaying the appropriate information for the currently active player.

Methods contained in the GameScreen Class that display dialogs, such as `showContextMenu()`, `promptEndTurn()` and `promptQuit()` will need to be tested on a live version of the software in order to make sure that the correct dialogs are displayed when the appropriate action is performed by the user. Once these dialog boxes have appeared, the test must confirm that each can be interacted with in the correct manner, and then perform any intended operations, such as quitting the game.

2.3.2 Flow Component Testing Interface

2.3.2.1 Flow Component Description

The Flow component (see **Section 2.2** of the design document) holds the responsibility for gathering the Player input (primarily mouse clicks) and forwarding it to the Board if it had gameplay ramifications, and to the Display if it did not. The Flow component is in charge of directing the flow of the system as it parses the input from the user, and determines what to do as a result of that input. The Flow component is also responsible for storing and handling specific system data. Again, from an MVC perspective, the Event Handler could be partly View in that it interacts with the outside users of the system, but it can also be partly Controller, as it has to know where to send events.

The Flow component will be tested to determine that it receives the correct input and responds with the correct operation. Additionally, the Flow component's saving, loading, and handling of data will be

tested to determine that the data will not be lost or manipulated incorrectly. As with elements in the Display component above, pieces of the Flow component generally cannot be handled with automatic unit testing, and will require a human user to provide input and verify the result of each interaction.

2.3.2.2 Significance of the Flow

The Flow component is significant because it receives and handles the input from the user, and then informs the other components, the Board and Display, of the user's intentions. If the Flow component malfunctions, the user will not be able to interact with the system and the system will fail. Also, since the Flow component saves, loads, and manipulates much of the data, it must function correctly, or the user will not be able to use the system as it is intended to be used. Malfunctions such as the inability to save game settings, player settings, or disabling the saving of game log files must be avoided.

2.3.2.3 GameManager Class Testing Plan

The GameManager class handles all the interactions between the player and the screen during a game. It is responsible for determining the action the player has taken and then directing the Board and Display components to respond accordingly.

The most important method of the GameManager class that must be tested for accuracy is its constructor. The GameManager takes in some parameters for a match, such as the number of players, and creates a match. A human must verify that the GameManager, given certain game settings, is creating a match using the parameters given.

A human must also carefully check that the GameManager is catching important interactions within the game, such as mouse clicks and drags. Each button or UI element must perform the correct action after its instantiation.

2.3.2.4 MenuManager Class Testing Plan

The MenuManager class helps with the user inputs given on all screens that interface with the user. In other words, all events in all the screens, except for the game screen, are reported to the MenuManager class. As a result, the MenuManager class stores information about the match options and the general settings.

The MenuManager class will be tested to determine that it receives input and saves and loads the stored data, such as player settings and robot records, as necessary. As with the GameManager, it also needs a human to interact with UI elements, such as buttons, in order to make sure that the correct actions are being performed with the corresponding interaction.

2.3.3 Logger Component Testing Interface

2.3.3.1 Logger Component Description

The Logger (see **Section 2.6** of the design document) is responsible for holding textual information (logs) of the game progress and actions. Each team possesses their own independent Logger that only stores the progress and actions known to the particular team that owns it. The Logger logs game piece movement, shots, and damage dealt and taken among other information that is visible to the particular team that owns the Logger. Also, the different logged information is categorized and stored

independently to allow the user the freedom to display some subset of the logged information he or she is interested in, all of the logged information, or none of the logged information.

2.3.3.2 Significance of the Logger

The Logger is essential for allowing the user to analyze the past actions of their opponents that are visible to them as well as their own past actions. Some players may like to analyze play history to learn from their own mistakes or their opponents past plays. The Logger is also important for storing logs of every action that takes place within a game, which allows the user to reconstruct past games. Essentially, the Logger is a highly desired feature and is important for correctly displaying and storing logs of the game play as promised within the game screen.

2.3.3.3 Entry Class Testing Plan

Unlike the Display and Flow components, some aspects of the Logger component can be tested with unit tests. Mock events can be introduced into the system in order to test that the Logger is responding to input correctly. But once the Logger has been integrated into the system, a human tester should verify that events within a match are being logged correctly. The Logger is not responsible for verifying whether the information it is being passed is valid, only for correctly logging the information it has been passed.

The Entry class is the smallest piece of the Logger component. It describes a single event that occurred within a match, including the type of event and who it was performed on. The Test cases for the Entry class are below.

- `setMovement(TeamEnum team, PieceEnum piece, HexCoord start, HexCoord end)`:
    - **Test cases: The `setMovement()` method is called with various teams, pieces, and starting and ending spaces.**
      Expected result: The entry is formatted as a `MOVEMENT` entry and stored correctly in the entry attribute. An example format is: “Blue Scout moves from (a, b, c) to (x, y, z).”
- `setShooting(TeamEnum team, PieceEnum piece, HexCoord target)`:
    - **Test cases: The `setShooting()` method is called with various teams, pieces, and starting and ending spaces.**
      Expected result: The entry is formatted as a `SHOOTING` entry and stored correctly in the entry attribute. An example format is: “Red Scout shoots (x, y, z), kills Blue Scout, and damages Green Tank (Health: 3 -> 2).”
- `setDamaged(TeamEnum team, PieceEnum piece, Integer damage)`:
    - **Test cases: The `setDamaged()` method is called with various teams, pieces, and starting and ending spaces.**
      Expected result: The entry is formatted as a `DAMAGED` entry and stored correctly in the entry attribute. An example format is: “Blue Tank received 2 point(s) damage from Red Sniper.”

2.3.3.4 Log Class Testing Plan

Built upon the Entry class, the Log class contains a list of Entry objects, cataloguing the match from the perspective of a single team (or from the perspective of an all-seeing spectator). The Log Test cases are listed below.

- `isGameOver()`:
    - **Test case: `isGameOver()` is called when all the entries in the `isPieceAlive` array are `true`.**
      Expected result: `isGameOver()` returns `false`.
    - **Test case: `isGameOver()` is called when all but one of the entries in the `isPieceAlive` array are `true`.**
      Expected result: `isGameOver()` returns `false`.
    - **Test case: `isGameOver()` is called when all but two of the entries in the `isPieceAlive` array are `true`.**
      Expected result: `isGameOver()` returns `false`.
    - **Test case: `isGameOver()` is called when all the entries in the `isPieceAlive` array are `false`.**
      Expected result: `isGameOver()` returns `true`.

- `addEntry()`:
    - **Test case: A `MOVEMENT` entry is added to the Log.**
      Expected result: The entry is of type `MOVEMENT` and is formatted correctly. The entry should also be placed into the `teamLog` list and the `movementEntries` list.
    - **Test case: A `SHOOTING` entry is added to the Log.**
      Expected result: The entry is of type `MOVEMENT` and is formatted correctly. The entry should also be placed into the `teamLog` list and the `shootingEntries` list.
    - **Test case: A `DAMAGED` entry is added to the Log.**
      Expected result: The entry is of type `DAMAGED` and is formatted correctly. The entry should also be placed into the `teamLog` list and the `damagedEntries` list.

2.3.3.5 Logger Class Testing Plan

Finally, the Logger class incorporates up to 7 Logs, in order to store the match from the perspectives of each team, plus the all-seeing spectator. This eliminates the need to search for the entries related to a particular team. The Logger Test cases are listed below.

- `clear()`:
    - **Test case: Set up a series of logged events of various types from various teams and pieces, then execute the `clear()` method.**
      Expected result: The entire collection of `teamLog` has been cleared.

3.0 Updates on Requirements and Design Documentation

Just as in the Design Document, there have been some changes to the previous documents based on lessons learned and clarifications since their completion. Below are several changes to previous documents.

3.1 Removal of Asynchronous Network Option from Requirements

Going through the testing interfaces, it was realized that creating a networked game creates a lot of failure modes—and failure modes that likely depend on external code bases—that would need to be tested. It was estimated that testing any implementation of network play for the game would require at least as much testing as the current list of all tests. Given a limited time budget for testing, the uncertain nature of the value asynchronous network play would create, and the lack of familiarity with network code construction, this option has been removed from the requirements.

3.2 Addition of Timeout to Interpreter Component

When creating the Interpreter test interface, a ubiquitous failure mode outside of the game's control is an AI script that does not execute properly due to incorrect use of the Forth language. While it is being considered to test each script at time of download for logic errors, a failsafe to ensure completion of the game, even if the game is not very engaging, is recommended. (Discussion with several veteran game players has shown that a game that 'hangs' or otherwise cannot run is considered a far worse error than a game that is not challenging.) To guard against this issue, a Timer will be added to the Interpreter Component which, if reached, will end a computer player's turn immediately. The value of the timeout is expected to be a match setting that is configurable by the Host Actor.

3.3 `spaceShotFrom` Parameter Added to `GameScreen.shootSpace()`

A much smaller concern is that a parameter for the GameScreen's `shootSpace()` method was left out of the method signature. This parameter, `spaceShotFrom`, represents the space from which the shot originates. This is for display purposes only.

3.4 `addMessage()` Method Added to Mailbox Class

Another small addition to the design document is the `addMessage()` method of the Mailbox class, which allows public access to the private `messages` field. It takes a single parameter, a String, and appends it to the `messages` field.

4.0 Summary

Testing is the keystone of strategies that ensure a software system behaves correctly. Rigorous testing of individual components of a system can prevent costly errors that detract from the user's experience. For this project, the development team's primary strategy for ensuring correctness in the system is unit testing each component. The two most important components that must be validated are the Interpreter and Board components.

The Interpreter must be handled carefully because it contains user-created input in the form of robot programs. User-created content should be validated to make sure that if it is malicious or malformed, it cannot damage the system as a whole. Each of the operations in the Interpreter will be tested so that they not only produce the correct answer when given correct input, but fail safely when given incorrect

input. The team has prioritized user experience over intelligent computer opponents, and so the blanket policy for Interpreter failures is that any user program error will cause the AI's turn to end immediately.

The Board component is important because of its centrality to the game system. Many other components communicate with the Board, which is responsible for containing and updating the game state. It tracks the game board and where each of the pieces are on it, and so unit testing is required to make sure that it is performing these operations correctly. If the Board is malfunctioning in some way, the game cannot be played correctly and a match might end prematurely, or not end at all.

Testing other components in the system is less vital, but still valuable at moments unit tests are able to be constructed. Some components, like the Display and Flow, rely heavily on user input and screen displays, and have elements that cannot be validated automatically with tests. These components will be tested where possible, but much of their correctness comes from testing the constructed system on human users, which is outside the scope of this document.