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. Aspects will be incorporated for a much stronger testing infrastructure if time permits. However, this is not a firm commitment for this system.

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 testing.

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

[Executive Summary 2](#_Toc466164846)

[1.0 Introduction 3](#_Toc466164847)

[1.1 Primary Test Pattern: SPVM Unit Testing 3](#_Toc466164848)

[1.1.1 Justification of SPVM Unit Testing 3](#_Toc466164849)

[1.2 Optional Test Pattern: Aspects as an Integration Test Pattern 3](#_Toc466164850)

[2.0 Testing Interfaces 3](#_Toc466164851)

[2.1 Interpreter Component Testing Interface 3](#_Toc466164852)

[2.1.1 Interpreter Component Description 3](#_Toc466164853)

[2.1.2 Significance of the Interpreter 4](#_Toc466164854)

[2.1.3 Interpreter Class Testing Plan 4](#_Toc466164855)

[2.1.4 AI Class Testing Plan 10](#_Toc466164856)

[2.1.5 Mailbox Class Testing Plan 11](#_Toc466164857)

[2.2 Board Component Testing Interface 12](#_Toc466164858)

[2.3 Flow Component Testing Interface 23](#_Toc466164859)

[2.4 Logger Component Testing Interface 28](#_Toc466164860)

[3.0 Updates on Requirements and Design Documentation 30](#_Toc466164861)

[3.1 Removal of Asynchronous Network Option from Requirements 30](#_Toc466164862)

[3.2 Addition of Timeout to Interpreter Component 31](#_Toc466164863)

[3.3 spaceShotFrom Parameter Added to GameScreen.shootSpace() Method 31](#_Toc466164864)

# 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, another testing paradigm will be examined, 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.

### 1.1.1 Justification of SPVM Unit Testing

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.

## 1.2 Optional Test Pattern: Aspects as an Integration Test Pattern

There is always a risk that a concentration on unit testing will cause issues in integration of components. While it is not expected to be a major risk, if the risk does materialize aspect tests will be used to determine integration concerns.

Aspects are excellent for integration testing, as aspects can create tests on events and other triggers among a variety of components. Checking at these crosscuts becomes as straightforward as unit tests using SPVM or another unit testing pattern.

# Testing Interfaces

## 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 was 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() :
  + 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() :
  + 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() :
  + 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 pieceIDs 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 pieceIDs of the messages it contains.  
    Expected result: hasMessage() returns true.
* receiveMessage():
  + 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 pieceIDs 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 pieceIDs 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 pieceIDs 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():
  + 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

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 and ensuring that each play is valid according to the game rules, and that the change to the state of the board is propogated 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 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. These tests includes testing the correct piece is moved, pieces not occupying a shot space remain undamaged, the next active piece is alive, only spaces within range are scanned, converting to one representation and back gives the correct result, and testing spaces just within bounds, the bounds themselves, and spaces considered to be out of bounds. 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.

Board Class tests:

Description:

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, and 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 while also checking that each action is valid. 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.

Significance:

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 and correct, other components receive correct updates, the flow of the game is directed correctly and smoothly, and the game rules are adhered to.

Test Cases:

* 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.

* + Test Case: Perform two valid move operations follow by a third on the same piece.

Expected Results: The first two moves are performed correctly, but the third is not and the error is handled.

* 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 piece’s 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 posses 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 the error is handled.

* + Test Case: The coordinate represents a valid coordinate outside bounds.
  + Expected Results: The scan is not performed and the error is handled.
  + Test Case: The coordinate represents an invalid coordinate.

Expected Results: The scan is not performed and the error is handled.

* + Test Case: Perform a valid scan.

Expected Results: The scan returns coordinates that are occupied.

* + Test Case: Perform a valid scan.

Expected Results: The scan does not return coordinates that are not occupied.

* + Test Case: Perform a valid scan.

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 the error handled.

* + Test Case: The coordinate represents an invalid coordinate.

Expected Results: The scan is not performed and the error handled.

* + 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 only all 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 the error handled.

* + Test Case: The coordinate represents an invalid coordinate outside bounds.

Expected Results: The conversion is not performed and the error handled.

* + 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 cooordinate represents a valid coordinate within bounds.

Expected Result: Returns true.

* + Test Case: The cooordinate represents a valid coordinate outside bounds.

Expected Result: Returns false.

* + Test Case: The cooordinate represents an invalid coordinate.

Expected Result: The error is handled.

* PieceEnum:
  + Each enumeration correctly represents a unique piece.
  + Each piece is enumerated.
  + The enumerations are well defined.

HexCoord Class tests:

Description:

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 visual display of a game piece.

Significance:

The HexCoord class is significant because it represents the coordinates of the hexagon spaces on the game board, but moreso 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.

Test Cases:

* Setters and Getters:
  + Test Case: Perform each getter method.

Expected Result: Each getter returns its correct value.

* + Test Case: Perform each setter method.

Expected Result: Each setter sets its respective value.

* reduce():
  + Test Case: Peform the reduce method on a coordinate representing the center.

Expected Result: The center is correctly reduced.

* + Test Case: Peform the reduce method on coordinates just beyond the center.

Expected Result: The coordinates are correctly reduced.

* + Test Case: Peform the reduce method on coordinates between the center and boundaries.

Expected Result: The coordinates are correctly reduced.

* + Test Case: Peform the reduce method on coordinates just before the boundaries.

Expected Result: The coordinates are correctly reduced.

* + Test Case: Peform the reduce method the boundary coordinates.

Expected Result: The cpordinates are correctly reduced.

* + Test Case: Peform the reduce method on a coordinate outside the boundries.

Expected Result: The coordinate is not reduced and the error handled.

Display Component Tests:

Description:

The Display component 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, along with the Flow, would be the View component of an MVC architecture. Even more than the Pieces component which appears next, the Display is a passive component and does not send information out to other components.

The Display component will be tested to determine that it correctly initializes the various screens, correctly switches between screens, that each screen is effective and accurate, and that each of the screen buttons performs their intended operation.

Display Class tests:

Description:

The Display class is designed to model the Display component. The Display class stores an array of all the screens, which can be easily indexed using the ScreenEnum enumeration that is assigned to each of the screens of the Screen class. The public switchToScreen() method takes in the enumeration of the screen to switch to and then updates the Display component to display the specified screen. The private initializer methods are used to initialize the various screens held by the Display. The Display also provides an accessor to retrieve the GameScreen to allow easier updating of the GameScreen.

Significance:

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.

Test Cases:

* Display(Logger logger) and initializer methods for each Screen:
  + Test Case: Inititialize the Display when the logger is null.

Expected Result: The Display is not initialized and the error handled.

* + Test Case: Inititialize the Display when the logger is not null.

Expected Result: The Display is initialized smoothly and correctly.

* + Test Case: Initialize each Screen with its respective initializer method.

Expected Result: The Screen is initialized correctly and smoothly.

* + Test Case: View each Screen after initialization.

Expected Result: Each Screen is appealing and conveys its data well.

* + Test Case: Click the buttons of each Screen.

Expected Result: Each button performs the intended operation.

* switchTo(ScreenEnum screen):
  + Test Case: screen does not represent a valid screen.

Expected Result: The Display does not change and the error handled.

* + Test Case: screen represents a valid screen.

Expected Result: The Display changes to the correct screen.

* + Test Case: Peform the method for each enumeration with a time delay between invocations.

Expected Result: The Display switchs to each Screen in the correct sequence with the correct delay.

* getGameScreen():
  + Test Case: A game is in progress.

Expected Result: The game screen is returned.

* + Test Case: A game is not in progress.

Expected Result: Null is returned.

* ScreenEnum:
  + Each enumeration correctly represents a unique screen.
  + Each Screen is enumerated.
  + The enumerations are well defined.

Screen Class tests:

Description:

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 component. 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, and will inherit from JPanel and holds all of the display elements for that screen, including buttons, images, and other control widgets.

Significance:

The Screen class is significant because it defines the basis of each screen. The Screen class must work correctly and guarantee to be displayed or no Screen within the Display component will be displayed correctly.

Test Cases:

* Screen(ScreenEnum screen)
  + Test Case: Perform the initialization.

Expected Result: The Screen is correctly initialized with the correct enumeration.

* getEnum():
  + Test Case: Perform the method for each Screen.

Expected Result: The correct enumeration for each Screen is returned.

GameScreen Class tests:

Description:

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 be stored.

Significance:

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 and effectively. If the game is not visualized correctly or effectively, the system will not succeed.

Test Cases:

* GameScreen():
  + Test Case: Perform the construction for 2, 3, and 6 teams.

Expected Result: The necessary images are initialized.

* + Test Case: Perform the construction for 2, 3, and 6 teams.

Expected Result: The teams-bar is correctly positioned an initialized with the correct ordering and coloring of teams.

* + Test Case: Perform the construction for 2, 3, and 6 teams.

Expected Result: The log screen is correctly initialized and positioned.

* + Test Case: Perform the construction for 2, 3, and 6 teams.

Expected Result: The context menu screen is correctly initialized.

* + Test Case: Perform the construction for 2, 3, and 6 teams.

Expected Result: The game board is correctly initialized and positioned.

* + Test Case: Perform the construction for 2, 3, and 6 teams.

Expected Result: The game piece images are correctly initialized and positioned.

* + Test Case: Perform the construction for 2, 3, and 6 teams.

Expected Result: The side buttons are correctly initialized and positioned.

* + Test Case: Perform the construction for 2, 3, and 6 teams.

Expected Result: The end turn button is correctly initialized and positioned.

* + Test Case: Perform the construction for 2, 3, and 6 teams.

Expected Result: The quit button is correctly initialized and positioned.

* + Test Case: Perform click each button on the GameScreen.

Expected Result: Each button performs its intended operation.

* update():
  + Test Case: Change the location of a piece.

Expected Result: The piece is updated to the correct position.

* + Test Case: Pan the board.

Expected Result: The board is correctly panned.

* + Test Case: Switch the active team and piece.

Expected Result: The top teams-bar, active piece, board focus, and board visibility is updated correctly.

* movePiece(PieceEnum piece, HexCoord coord):
  + Test Case: A valid move is performed.

Expected Result: Only the piece parameter is visually moved to the coordinate and no other pieces are moved.

* + Test Case: The coordinate represents a valid hexagon coordinate within bounds.

Expected Result: The piece is visually moved to the coordinate.

* + Test Case: The coordinate represents a valid hexagon coordinate outside bounds.

Expected Result: The piece is not visually moved and the error handled.

* + Test Case: The coordinate represents an invalid hexagon coordinate.

Expected Result: The piece is not visually moved and the error handled.

* shootSpace(PieceEnum piece, HexCoord coord):
  + Test Case: A valid shot is performed.

Expected Result: Only the piece parameter shoots the coordinate and no other shots occur.

* + Test Case: The coordinate represents a valid hexagon coordinate within bounds.

Expected Result: A shot is visually fired from the piece to the coordinate.

* + Test Case: The coordinate represents a valid hexagon coordinate outside bounds.

Expected Result: A shot is not visually fired and the error handled.

* + Test Case: The coordinate represents an invalid hexagon coordinate.

Expected Result: A shot is not visually fired and the error handled.

* switchPlayer(PlayerEnum player, PieceEnum piece):
  + Test Case: Perform a valid switch.

Expected Result: The GameScreen is blacked-out, initializes itself for the new player, prompts the player to begin their turn, and the GameScreen is correct when the player acknowledges its their turn.

* + Test Case: The piece parameter belongs to the team parameter.

Expected Result: The switch visually occurs correctly.

* + Test Case: The piece parameter does not belong to the team parameter.

Expected Result: The switch does not occur and the error handled.

* + Test Case: Perform a valid switch for a live team.

Expected Result: The top teams-bar is updated correctly.

* + Test Case: Perform a valid switch for a dead team.

Expected Result: The GameScreen is not updated and the error handled.

* + Test Case: Perform a valid switch for a live piece.

Expected Result: The game board focus and visibility is updated correctly.

* + Test Case: Perform a valid switch for a dead piece.

Expected Result: The GameScreen does not update and the error handled.

* + Test Case: Perform a valid switch for a live piece and dead piece.

Expected Result: The game board and piece are positioned correctly.

* + Test Case: Perform a valid switch.

Expected Result: The game board and active piece are positioned correctly.

* + Test Case: Perform a valid switch.

Expected Result: The log is correctly updated for the new team.

* + Test Case: Perform an invalid switch.

Expected Result: The log is not updated and the error handled.

* updateLog(TeamEnum team):
  + Test Case: The team is a correct team.

Expected Result: The log is updated for the correct team.

* + Test Case: The team is not a correct team.

Expected Result: The log is not updated and the error is handled.

* + Test Case: Perform a valid log.

Expected Result: The log data belongs to the team.

* pan(int deltaX, int deltaY):
  + Test Case: The deltaX and deltaY keep the board position within the limits.

Expected Result: The pan is correctly performed according to the changes.

* + Test Case: The deltaX and deltaY push the board position outside the limits.

Expected Result: The pan is correctly performed only up to the position limits.

* + Test Case: Perform a valid pan.

Expected Result: The change in the board’s visual position remains after panning.

* showContextMenu(), promptEndTurn(), promptQuit(), and showTransition():
  + Test Case: Invoke showContextMenu().

Expected Result: The context menu is shown, sends the correct event upon a choice being clicked, and disappears.

* + Test Case: Invoke promptEndTurn().

Expected Result: The end turn prompt is shown, sends the correct event upon a choice being clicked, and disappears.

* + Test Case: Invoke promptQuit().

Expected Result: The quit game prompt is shown, sends the correct event upon a choice being clicked, and disappears.

* + Test Case: Invoke showTransition().

Expected Result: The GameScreen turns black for the transition and the prompt prompting the next team to acknowledge it is their turn appears and disappears after the team acknowledges.

* + Test Case: Click each button.

Expected Result: Each button performs its intended operation.

Proposed changes:

## 2.3 Flow Component Testing Interface

Flow Component Testing:

Description:

The Flow component holds the responsibility for gathering the Player input—using 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 partially in charge of directing the flow of the system as it parses the input from the user, who ultimately controls the flow, and determines what to do as a result of that input. Input will be received from the user in the form of mouse and button clicks, and the Flow component determines what action to take based on the specific event. Also, the Flow component is 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. It will be tested to ensure that when a specific event is received it performs the correct operation. Also, the Flow component’s saving, loading, and handling of data will be tested to determine the data will not be lost or manipulated incorrectly.

Significance:

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.

GameManager Class:

Description:

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.

Significance:

The GameManager is significant because it receives the events from each player during the game and directs the Board and Display components accordingly. If the GameManager malfunctions, the game will not run smoothly and could break, which would defeat the purpose of the system, which then leads to system failure.

Test Cases:

* GameManager(Settings settings):
  + Test Case: Instantiate the GameManager.

Expected Result: The buttons and the settings are instantiated correctly and can be accessed as necessary.

* createMatch(MatchOptions matchOptions):
  + Test Case: Instantiate a valid match.

Expected Result: The match is instantiated according to the specified match options. Check that each of the match options are valid and can be accessed as needed.

* + Test Case: Instantiate an invalid match.

Expected Result: The match is not instantiated and the error is handled appropriately.

* clickedButton():
  + Test Case: Trigger a button event.

Expected Result: Returns true.

* + Test Case: Trigger events not produced by the button.

Expected Result: Returns false.

* isValidButton():
  + Test Case: Trigger a button event generated from one of the buttons in the game screen.

Expected Result: Returns true.

* + Test Case: Trigger a button event outside the game screen.

Expected Result: Returns false.

* draggedBoard():
  + Test Case: Trigger an event generated by dragging the board.

Expected Result: Returns true.

* + Test Case: Trigger events not generated by dragging the board.

Expected Result: Returns false.

* clickedMove():
  + Test Case: Trigger an event generated by clicking move.

Expected Result: Returns true.

* + Test Case: Trigger events not generated by clicking move.

Expected Result: Returns false.

* clickedShoot():
  + Test Case: Trigger an event generated by clicking shoot.

Expected Result: Returns true.

* + Test Case: Trigger events not generated by clicking shoot.

Expected Result: Returns false.

* clickedExit():
  + Test Case: Trigger an event generated by clicking the exit button.

Expected Result: Returns true.

* + Test Case: Trigger events not generated by clicking the exit button.

Expected Result: Returns false.

* clickedPanArrow():
  + Test Case: Trigger an event generated by clicking the pan arrow button.

Expected Result: Returns true.

* + Test Case: Trigger an events not generated by clicking the pan arrow button.

Expected Result: Returns false.

MenuManager Class:

Description:

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.

Significance:

The MenuManager class stores and loads data the system needs to properly run the game and receives input about the events of screens other than the game screen. Since this data is needed to properly run the game, it is necessary to ensure that this data can be saved and loaded successfully on each run of the system.

Test Cases:

* saveSettings():
  + Test Case: Set valid settings and invoke the method.
  + Expected Result: The settings are saved in the correct format according to the current state of the settings.
  + Test Case: Set invalid settings and invoke the method.

Expected Result: The settings are not saved and the error is handled appropriately.

* LoadSettings():
  + Test Case: Store valid settings and invoke the method.

Expected Result: The settings are loaded into the system successfully and correctly.

* + Test Case: Store invalid settings and invoke the method.
  + Expected Result: The settings are not loaded into the system and the error is handled appropriately.
* clickedButton():
  + Test Case: Trigger a button event.

Expected Result: Returns true.

* + Test Case: Trigger events not produced by the button.

Expected Result: Returns false.

* isValidButton():
  + Test Case: Trigger a button event generated from one of the buttons in the game screen.

Expected Result: Returns true.

* + Test Case: Trigger a button event outside the game screen.

Expected Result: Returns false.

PlayerSettings Class:

Description:

The PlayerSettings class contains information about the preferences of the player while playing the game. It is also used to instantiate each player before the game begins. Each player will be instantiated according to their player settings.

Significance:

The PlayerSettingsClass is responsible for storing the preferences of each player. Since it is used to instantiate the game with the preferences for each player, it is important to ensure correct retrieval and storage of each player’s preferences so that a game may begin successfully.

Test Cases:

* PlayerSettings(bool isAI, String name, Dictionary<int, RobotRecord> mapRobots):
  + Test Case: Instantiate the object with various values for isAI, name, and mapRobots.

Expected Results: Check that all fields are set correctly and can be accessed as needed.

* getIsAI():
  + Test Case: Set isAI of the object to true.

Expected Results: Returns true.

* + Test Case: Set isAI of the object to false.

Expected Results: Returns false.

* setIsAI(bool isAI):
  + Test Case: Use to set isAI of the object to true.

Expected Results: The isAI field is true.

* + Test Case: Use to set isAI field of the object to false.

Expected Results: The isAI field is false.

* getName():
  + Test Case: Set name field to various names.

Expected Results: .Returns the correct name.

* setName(String name):
  + Test Case: Use to set name field of the object to true.

Expected Results: The name field is altered to the name parameter.

Settings Class:  
Description:

The Settings class stores the information that the user would like to store about the future match. For example, the volume of the background sound during the match, whether or not the logs should be stored to the disk, and additional settings, such as keyboard inputs, that the user would like to use for movement and direction of the pieces.

Significance:

The options for the current and future matches are stored within the Settings class, so it is important to ensure this data can be stored and accessed for a game to be instantiated properly and as the user desires.

Test Cases:

* Settings(int soundVolume, bool saveLogs, Dictionary<String, int> keyMapping):
  + Test Case: Instantiate the object with various values for soundVolume, saveLogs, and keyMapping.

Expected Results: Check that all fields are set correctly and can be accessed as needed.

* setVolumeLevel(int level):
  + Test Case: Use to set the soundVolume field to various valid levels.

Expected Results: The soundVolume field is set to the specified level.

* + Test Case: Use to set the soundVolume field to the minimum level.

Expected Results: The soundVolume field is set to the specified level.

* + Test Case: Use to set the soundVolume field to a level below the minimum.

Expected Results: The soundVolume field is set to the minimum level.

* + Test Case: Use to set the soundVolume field to the maximum level.

Expected Results: The soundVolume field is set to the specified level.

* + Test Case: Use to set the soundVolume field to a level above the maximum level.

Expected Results: The soundVolume field is set to the maximum level.

* getVolumeLevel():
  + Test Case: Set the soundVolume field to various different levels.

Expected Results: The correct sound level is returned.

* setSaveToDisk(bool save):
  + Test Case: Use to set the saveLogsToDisk field to true.

Expected Results: The saveLogsToDisk field is true.

* + Test Case: Use to set the saveLogsToDisk field to false.

Expected Results: The saveLogsToDisk field is false.

* getSaveToDisk():
  + Test Case: Set the saveLogsToDisk field to true.

Expected Results: Returns true.

* + Test Case: Set the saveLogsToDisk field to false.

Expected Results: Returns false.

MatchOptions Class:

Description:

The MatchOptions class is essentially for storing all the details about a match, if in case the player wants to rematch. The MatchOptions is not resposible for ensuring the data is correct. This is already done elsewhere.

The MatchOptions class does not contain many methods other than accessors and mutators, but merely acts as a temporary storage. As such, it will only be tested to determine that it correctly stores the data and that the data can then be retrieved as needed.

Significance:

The MatchOptions class is significant because it stores the match options specified by the user, which are needed if the user would like to play a rematch. It is important to ensure the MatchOptions class stores the options for the match so that this feature can be properly implemented.

TestCases:

* Constructor:
  + Test Case: Instantiate the object with various values for its attributes.

Expected Results: The values of its attributes are stored correctly and can be accessed as needed.

* Setters and Getters:
  + Test Case: Use setters to set the attributes to various values.

Expected Results: The attributes are set the specified value.

* + Test Case: Set the attributes to various values and use the getters.

Expected Results: The getters return the correct values.

RobotRecord Class:

Description:

The RobotRecord class has the sole purpose of storing the JSON file.

The RobotRecord class will be tested to ensure it correcly stores the JSON file and the JSON file can then be accessed correctly.

Significance:

The RobotRecord is significant because it is used to store the JSON files of the robots. As such, it must be ensured that the JSON files are stored correctly and can then be accessed as needed, otherwise AI robots cannot play in the game, which is an important feature.

Test Cases:

* RobotRecord(JSON file):
  + Test Case: Instantiate the object with a valid JSON robot file.

Expected Results: The JSON file is stored correctly and can be accessed as needed.

## 2.4 Logger Component Testing Interface

Description:

Logger is one of the six components, and it holds textual information of the game progress. Each team has a different logger displayed according to its view scope so far, and the AI has all the loggers to help make intelligent decisions. It can show movement, shooting and damaged information separatedly, so the user has the freedom to choose only to display the part of information he or she is interested in, or all of them or none of them.

Significance:

Logger is not very important to every user. Some poeple like to analyze playing history to learn from their lessons, then Logger can help with this. To other people, logger records the game process, and it is the place to check the damaged interactions between teams, the movements of the pieces in the team and if you can continue to play.

// We test Entry Class first since it is the base

Entry Class

addEntry():

- 1 test for movement, compare entry field with expected string

- 1 test for shooting, compare entry field with expected string

- 1 test for damaged, compare entry field with expected string

setMovement():

getMovement():

- 2 tests for this setter and getter

check if the returned string from getter is right

(expected: start and end locations of one robot in one team)

(e.g. Scout moves from (a, b, c) to (x, y, z). )

setShooting():

getShooting():

- 2 tests for this setter and getter

check if the returned string from getter is right

(expected: shooting of one robot in one team)

(e.g. Scout shoots (x, y, z), kills Blue Team Scout and damages Green Team Tank (Health: 3 -> 2)! )

setDamaged():

getDamage():

- 2 tests for this setter and getter

check if the returned string from getter is right

(expected: damaged of one robot in one team)

(e.g. Your Tank get 1 point damage from Red Team. )

(e.g. Your Scout is killed by Red Team. )

// Then we test Log Class that contains entries for each team

Log Class

getTeamLog():

- 2 tests to check if the returned string is right

(expected: all logs of this team)

getTeamMovement():

- 2 tests to check if the returned string is right

(expected: oonly the movement logs of one team)

getTeamShooting():

- 2 tests to check if the returned string is right

(expected: only the shooting logs of one team)

getTeamDamaged():

- 2 tests to check if the returned string is right

(expected: only the damaged logs of one team)

isGameOver():

- 1 test for all three indexs in the isPieceAlive array is false, which should return true

- 1 test for at least one index in the isPieceAlive array is true, which should return false

// Last we test Logger Class that contains logs for all teams.

Logger Class

displayLog():

- 2 tests for two arbitrary teams

(expected: display all the logs of one team)

displayMovement()

- 2 tests for two teams

(expected: display only the movement logs of one team)

displayShooting()

- 2 tests for two teams other than those in tests for displayMovement()

(expected: display only the shooting logs of one team)

displayDamaged(TeamEnum player)

- 2 tests for two teams other than those in tests for displayMovement() and displayShooting()

(expected: display only the damaged logs of one team)

clear()

- 1 test for if it can display an empty string

# 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() Method

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.

Board stuff

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.