### 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.