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](#_Toc466117034)

[1.0 Introduction 2](#_Toc466117035)

[1.1 Primary Test Pattern: SPVM Unit Testing 2](#_Toc466117036)

[2.0 Testing Interfaces 3](#_Toc466117037)

[2.1 The Interpreter Testing Interface 3](#_Toc466117038)

[2.2 The Board Testing Interface 3](#_Toc466117039)

[2.3 Other Testing Interfaces 3](#_Toc466117040)

[2.4 A Notional Aspect-Oriented Testing Structure 3](#_Toc466117041)

[3.0 Updates on Requirements and Design Documentation 3](#_Toc466117042)

# 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 The Interpreter Testing Interface

Interpreter

Word interface tests:

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: with zero values on the stack.  
Expected result: there will be zero values on the stack.

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: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where one or both are not integers.  
    Expected result: ??? Toss or replace?
  + Test case: two integers on the stack  
    Expected result: the top of the stack contains the correct result of the addition
* subtract() :
  + Test case: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where one or both are not integers.  
    Expected result: ??? Toss or replace?
  + Test case: two integers on the stack  
    Expected result: the top of the stack contains the correct result of the subtraction
* multiply() :
  + Test case: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values where one or both are not integers.  
    Expected result: ??? Toss or replace?
  + Test case: two integers on the stack  
    Expected result: the top of the stack contains the correct result of the multiplication
* divideRemain() :
  + Test case: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values where one or both are not integers.  
    Expected result: ??? Toss or replace?
  + Test case: two integers on the stack 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: two integers on the stack 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 contains the correct remainder of the division
* and() :
  + Test case: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values where one or both are not boolean values  
    Expected result: ??? Toss or replace?
  + Test case: two boolean values on the stack  
    Expected result: the top of the stack contains the correct result of the and operation as a boolean, true or false
* or() :
  + Test case: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values where one or both are not boolean values  
    Expected result: ??? Toss or replace?
  + Test case: two boolean values on the stack  
    Expected result: the top of the stack contains the correct result of the or operation as a boolean, true or false
* invert() :
  + Test case: one non-boolean value on the stack

Expected result: ??? Toss or replace?

* + Test case: one boolean value on the stack  
    Expected result: the top of the stack contains the opposite boolean value
* duplicate() :
  + Test case: one value of any type on the stack  
    Expected result: the top of the stack contains two of the original value
* drop() :
  + Test case: one value of any type on the stack  
    Expected result: the value has been removed from the stack, and nothing has been pushed to the stack
* swap() :
  + Test case: one value of any type on the stack  
    Expected result: the value has been replaced at the top of the stack
  + Test case: two values of any type on the stack  
    Expected result: the second value is on the top of the stack, and the first is just below it
* rotate() :
  + Test case: one value of any type on the stack  
    Expected result: the value has been replaced at the top of the stack
  + Test case: two values of any type on the stack  
    Expected result: the two values have been replaced at the top of the stack in their original order
  + Test case: three values of any type on the stack  
    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: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are not the same type  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are the same type but not comparable
  + Test case: 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: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are not the same type  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are the same type but not comparable
  + Test case: 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: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are not the same type  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are the same type but not comparable
  + Test case: 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: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are not the same type  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are the same type but not comparable
  + Test case: 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: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are not the same type  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are the same type but not comparable
  + Test case: 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: fewer than two values on the stack  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are not the same type  
    Expected result: ??? Toss or replace?
  + Test case: two values on the stack where both are the same type but not comparable
  + Test case: 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 value is replaced on the stack??
  + 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.

## 2.2 The Board Testing Interface

**Board Component tests:**

**Description:**

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.

**Conditions to test:**

* Test the constructor initializes the board correctly by checking that the state of the board is as expected.
  + The correct number of teams are initialized.
  + The correct number of pieces are initialized according to the number of teams and each team has the correct number and types of game pieces.
  + The Logger is initialized and present.
  + The Display is initialized and present.
* Test the movePiece(PieceEnum piece, HexCoord coord) method correctly moves the piece by simulating a move event and then checking the board state for various correct and incorrect values.
  + The correct movement only occurs when the coord parameter represents a valid hexagon coordinate within the game board boundaries.
  + The correct piece is moved and it is moved to the correct space when a piece is moved less than its remaining mobility points.
  + The correct piece is moved and it is moved to the correct space when a piece is moved exactly its remaining mobility points.
  + The correct piece is not moved when a piece tries to move more than its remaining mobility points.
  + The correct number of mobility points are subtracted from the moving piece’s mobility points.
  + No pieces other than the specified piece are moved.
  + A piece is not moved if it is the piece’s third movement within its current turn.
* Test the shootSpace(PieceEnum piece, HexCoord coord) method correctly shoots at a space by simulating a shoot event and then checking the state of the game for various correct and incorrect values.
  + All pieces occupying the shot space are dealt the correct damage.
  + Pieces not occupying the shot space remain undamaged.
  + Shooting only occurs when the coord parameter represents a valid hexagon coordinate within range of the specified piece’s range points and within the game board boundaries.
  + A piece is not allowed the chance to shoot a second time during its current turn.
* Test the nextPiece() method correctly determines the next piece for different numbers of players, turns, and rounds.
  + The next piece is correctly determined for at least two whole rounds.
  + A piece is skipped if it does not posses any remaining health points.
  + The next piece is correctly determined when there are 2, 3, and 6 teams.
* Test scanArea(PieceEnum piece, HexCoord coord) method correctly returns the list of coordinates within range of the scan for various scanning positions and ranges.
  + The area is only scanned if the coord parameter represents a valid hexagon coordinate within the game board boundaries.
  + The scanned area scans only spaces within range of the piece’s range points from the piece’s current position.
  + The scan result includes spaces occupied inside the range of the scan from the piece’s current position.
  + The scan result does not include unoccupied spaces within or not within the rane of the scan.
  + The scan result does not include spaces occupied outside the range of the scan from the piece’s current position.
* Test scanSpace(HexCoord coord) method correctly returns the list of robots occupying a space for various spaces containing various robots.
  + The correct space is scanned.
  + The space is only scanned if the coord parameter represents a valid hexagon coordinate within the game board boundaries.
  + An empty list is returned when the scanned space is empty.
  + A list containing the correct robot is returned when one robot occupies the space.
  + The correct list of robots is returned when more than one robot occupies the space.
* Test both the absoluteToRelative(HexCoord coord) and relativeToAbsolute(HexCoord coord) return the correct converted space and the other returns the original.
  + The conversion is completed only when the coord parameter represents a valid hexagon coordinate within the game board boundaries.
  + The former function returns the correct relative representation of the absolute representation for various valid hexagon coordinates.
  + The latter function returns the correct absolute representation of the relative representation for various valid hexagon coordinates.
  + The original absolute hexagon coordinate representation can be obtained by converting it to its relative representation and back to its absolute representation.
  + The original relative hexagon coordinate representation can be obtained by converting it to its absolute representation and back to its relative representation.
  + No conversion is done if the coord parameter is not a valid hexagon coordinate within the game board boundaries.
* Test isInBounds(HexCoord coord) returns true for various correct input and false for various incorrect input. Check edge cases.
  + Returns true when the cood parameter represents a valid hexagon coordinate within the game board boundaries.
  + Returns false when the coord parameter represents a valid hexagon coordinate outside the game board boundaries.
  + Returns false when the coord parameter does not represent a valid hexagon coordinate.
* Test the enumeration 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.

**Conditions to test:**

* Test the setters and getters for the coordinates.
  + Each setter correctly sets its respective values.
  + Each getter correctly gets its respective values.
* Test the reduce() method correctly reduces the coordinates for various inputs. Check edge cases.
  + The coordinate is reduced to its correct minimal representation for various valid hexagon coordinates.
  + Center coordinates are correctly reduced.
  + Coordinates just beyond the center are correctly reduced.
  + Coordinates between the center and edges of the game board are correctly reduced.
  + Coordinates just within the boundaries are correctly reduced.
  + Coordinates at the boundary edges are correctly reduced.
  + Coordinates outside the game board boundaries are not reduced.

**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.

**Conditions to test:**

* Test the constructor of the Display and the initializer methods for the other screens. Visually check that each screen is correct and that each of the buttons performs the correct action in various situations. Update each screen if improvements are necessary.
  + Each Screen is correctly initialized.
  + Each Screen conveys data and actions meaningfully.
  + The visualizations are easily interpreted.
  + Each button performs the intended operation when clicked.
* Test the switchTo(ScreenEnum screen) method correctly switches to each of the screens.
  + Switches to the correct screen for each possible value of screen.
  + Does not switch screens when an incorrect parameter is passed.
* Test the getGameScreen() correctly returns the game screen.
  + Correctly returns the game screen when a game is in progress.
  + Returns null when a game is not in progress.
* Test the ScreenEnum.
  + Each enumeration correctly represents a unique screen.
  + 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.

**Conditions to test:**

* Test the constructor correctly initializes the screen.
* Test the getEnum() returns the correct enum for various screens.
  + Test that each of the enumerations are returned for each of the various Screens.
  + The correct enumeration 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 the constructor correctly initializes the game screen and each of the initializer methods correctly initializes the screen.
  + The necessary images are initialized.
  + The teams-bar is correctly positioned and initialized for various numbers of teams with the correct coloring and ordering of teams.
  + The log screen is correctly initialized and positioned.
  + The context menu is correctly initialized.
  + The game board is correctly initialized and positioned.
  + The images of the pieces are correctly initialized and positioned.
  + The side buttons are correctly initialized and positioned.
  + The end turn button is correctly initialized and positioned.
  + The quit button is correctly initialized and positioned.
  + Each button performs the intended operation when clicked.
* Test the update() method correctly updates the game screen according to the current game state for various game states.
  + The Display is correctly updated to its current state.
* Test the movePiece(PieceEnum piece, HexCoord coord) and shootSpace(PieceEnum piece , HexCoord coord) methods peform the correct visualizations of the move and shoot actions for various inputs.
  + The actions are only performed if the coord parameter represents a valid hexagon coordinate.
  + The actions are not performed if the coord parameter represents a valid hexagon coordinate.
  + The correct piece is moved to the correct position for various pieces and spaces and the correct piece performs the shot for various pieces and spaces.
  + The correct space is shot for various space.
* Test the switchPlayer(PlayerEnum player, PieceEnum piece) method correctly switches to the correct player for multiple different players. Test for dead players. Test for different game states.
  + The action is only performed if the piece parameter belongs to the team parameter.
  + The GameScreen goes black, initializes itself for the new player, and prompts the player to begin their turn. The GameScreen returns when the player acknowledges it is their turn by clicking the button on the prompt.
  + The top teams-bar is correctly updated.
  + The game board visibility is updated correctly.
  + The game board and piece are positioned correctly.
  + The log is updated correctly for the new team.
* Test the updateLog(TeamEnum team) correctly updates the log for various different actions and players.
  + Updates the log to display the log data for the current team.
  + All correct logs are shown.
  + The log data of other players is not shown.
  + Log data that should not be known to the team is not shown.
* Test the pan(int deltaX, int deltaY) method correctly pans the board for various inputs.
  + The pan is only performed if the deltaX and deltaY parameters are within the limits.
  + The game board is correctly panned for various valid inputs.
  + The game board does not pan further than the limits of visibility.
  + The pan is set permanently.
* Test the showContextMenu(), promptEndTurn(), promptQuit(), and showTransition() perform correctly.
  + Each displays correctly and their buttons perform the intended operation.

## 2.3 Other Testing Interfaces

## 2.4 A Notional Aspect-Oriented Testing Structure

# 3.0 Updates on Requirements and Design Documentation