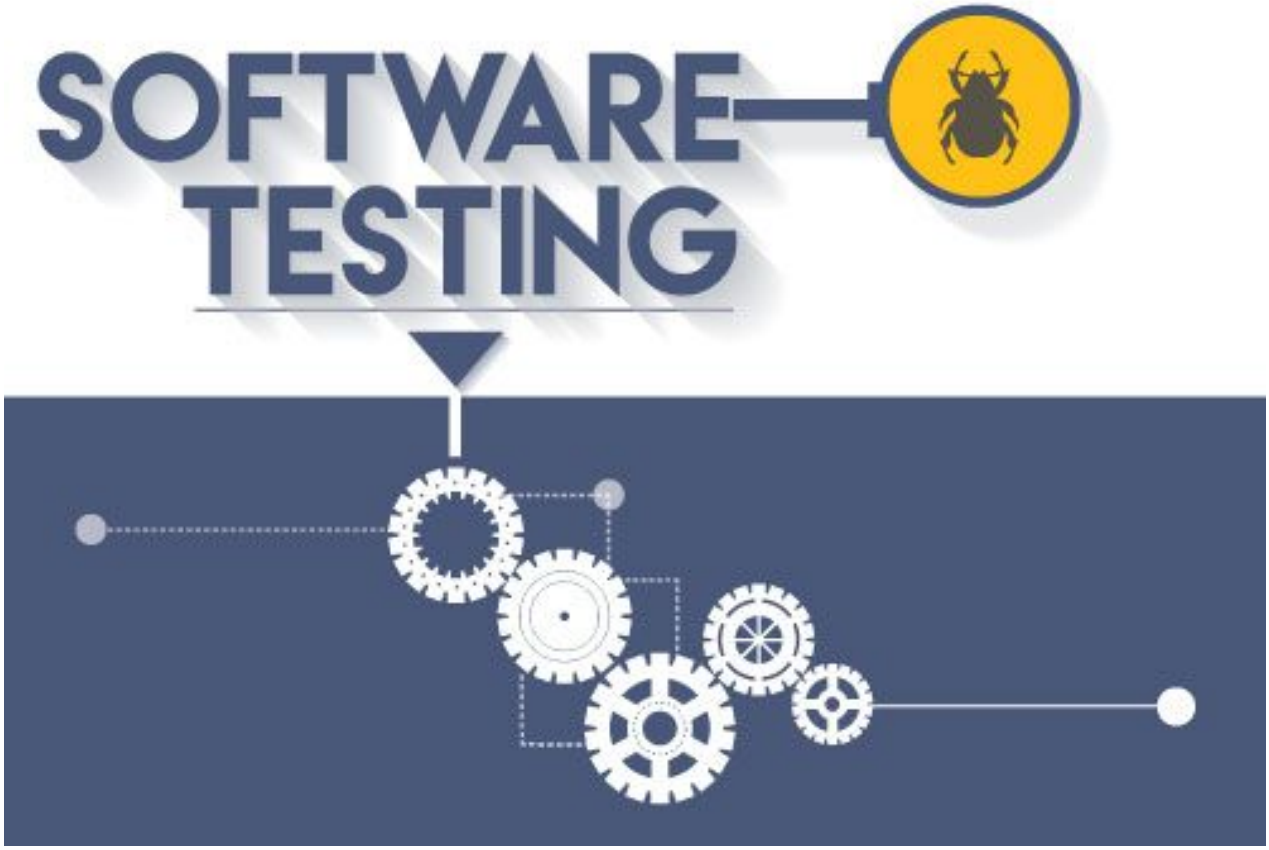

Hanabi Client

Group: C1

Table of Contents

1) Introduction

This document will describe all of the tests that we feel should be passed in order for us to consider our code correct. Tests will be conducted for end-to-end, integration, and unit cases. Some tests listed below are more general than others, while some contain more precise tests.

As we are developing, we will be conducting test driven development. This means that tests should be written prior to the start of programming. It is strongly encouraged that the programmers look throughout this document and find the tests that are listed for the section they are writing.

2) Primary Tests

2.1) End-to-end

I. "*Create Game*" use case:

1) Step 1: The user must have bash open and a wifi connection. In this instance, the player trying to create a game will not have a game created. They will enter the given bash command line with the options they choose. We must check that the game has successfully been created with the settings they inputted.

II. "*Join Game*" use case:

1) Step 1: The user must have bash open and a wifi connection. The user will enter the bash command to join a game which specifies which game they are going to join. If required, they will add a tag with the special tag given to you by the game manager. This test will check that the user has successfully joined the game (one possible way to check is we could check the ID's that are in the board class).

III. "*Human Play Card*" use case:

1) Step 1: This step will require the following things to begin: It must be our user's turn to select a move, and the user must not have clicked any buttons yet. The screen should first indicate to the player to select a move. After clicking the button that says 'play card', the screen will indicate to the player to select a card.
2) Step 2: The second step will require several things. The first will be that it is currently the

users turn. The second is that the user must have clicked on the button that says “play card,” and nothing else. If these requirements are met, the screen should indicate to the player to “Select a card”. The player will then click on their own card. After clicking on their card, the screen will indicate that it is no longer their turn. This test must check several things after the player has clicked their card. The first is that the user no longer has the card they selected in their hand. The second is that the user has a new card in their hand. The third is that the server received the card from user, and placed it in the correct position.

IV. “*Human Discard Card*” use case:

1) Step 1: For this step to occur it must be the user’s turn and the team must have less than seven information tokens. Following clicking the button that says “Discard a card,” the game should indicate the player to select a card to be discarded.
2) Step 2: This step will proceed as long as it is still the user’s turn and the team still has less than seven information tokens. The discard button also must have been clicked, with the screen prompting the user to select a card. After a card has been selected to be discarded, the tests will check that the user no longer has the discarded card in their hand and that they have gained a

new card. Lastly, the test will make sure you gained an information token from the discard.

V. "*Human Tell Information*" use case:

1) Step 1: This step will be started when it is the user's turn and the team still has at least one information token left. After clicking the "Tell information" button, several buttons will appear on the screen. Each representing a player. To provide information to a desired player the user must click on the button corresponding with said player.
2) Step 2: This step will proceed if it still the user's turn, the team has at least one token left, and that step one is done. A few buttons indicating colors and number will appear and clicking on the buttons indicates what information the user wants to tell to the player he has chosen. Once the desired card is chosen, the test will assert that the correct JSON message was outputted, which includes the correct chosen player and card.

VI. "*Human Turn on AI mode*" use case:

1) Step 1: This step will require that the user is in a game of Hanabi, and that it is the user's turn (IE, the current game state is 'READY'). Following clicking the button that says "AI mode," the game should indicate the users AI mode has been turned on completely. To check that it worked we will assert that the current state is now 'AI.'

2) Step 3: In order to test the process of regaining control from the AI, we must simply check controller states. In order to do this, we must first create a mock game of hanabi, where it is our players turn. We must then simulate a click on the AI mode button, which should cause the controller to change to the AI State. Following this, we must wait for the AI to send a JSON message, and assert that our controller goes into the WAIT state, and that this JSON message is of a normal format.

VII. "*AI Play Card*" use case:

1) Step 1: This step will require several things. First, it must be our client's turn to select a move, and our client must be an AI player. This step involves calculating the most probable card. A test for this could involve constructing a game in which the discard pile has no cards, and all other players in the game (besides our client) have cards, of any color, with numerical values between two and five (inclusive). This test should show that the AI calculated that the most probable card was a number 1 of any color.
2) Step 2: This test will require several things. The first is that it is our client's turn, and that our client is an AI player. Secondly, the AI player must have already calculated the most probable card. Thirdly, the AI player must want to play the card that it calculated. Given the same scenario as part one for this use case, if we additionally

enforce that there is no played cards, our AI player will have calculated the most probable card in it's hand is a number one of any color. Using this information, we can assure that the AI player will play its number one. This action should result in several things. The first is that the card in it's hand will be sent to the server as a JSON string. The second is that the card will no longer be in our player's hand. The final thing that must be true is that the player will have had to have drawn a new card.

VIII. "*AI Discard Card*" use case:

1) Step 1: This step will require several things. First, our client must be an AI player, and it is the turn that client can move, and AI team must have less than seven information tokens. A test for this could involve creating an instance of a hanabi game where the most probable card is a green 3, but the highest played green card is two. Given this scenario, this method should return a JSON message describing discarding a index of a card in our players hand.
2) Step 2: This test will require several things. Firstly, it must be the client's turn and the client must be an AI player. Secondly, the AI player must have already calculated the most likely card to discard. Thirdly, the AI player must want to discard the card that it calculated.

IX. “*AI Tell Information*” use case:

1) Step 1: This test will require two things before it can proceed. The first is that it must be the user’s turn. The second is that the team must have at least one information token. This test will require the AI to view every players hand. It will then look to see if any of its teammates have viable cards. If any of its teammates have viable cards, it must send a JSON message indicating that it is telling a player about a card in their hand to the server. For instance, consider a hypothetical game with three players. Player one is our user. Player two has cards only with numerical values greater than or equal to two. Player 3 only has cards with numerical values of one and the first card in their hand is red. There are seven information tokens available, and no cards have been played. In this described instance, this step should send a JSON message saying that it’s sending Player information about all of their cards with the numerical value one.
    - If a similar circumstance, but where player 3 knows that their card is the number one, our AI should send a JSON message to the server indicating that it wants to tell player three about the red cards in their hand.

X. “*See Discard Pile*” use case:

1) As long as a game of Hanabi is currently being played then the user can see the discard cards at any time. The user will click the “See Discard Pile” button, and the user will be given a textual representation of the cards that have been discarded throughout the game. If no cards have been discarded then the client will just return an empty string. To close the discard pile information the user clicks the show discard pile button again. The only test we can perform for this method is to visually witness the above interactions.

XI. “*See Stats*” use case:

1) Step 1: As long as a game of Hanabi is currently being played then the user can see which card is the most probable card in their hand at any time. The user will click the “See Stats” button, and the user will be given a textual representation the most probable card that is in our user’s hand. To close this information, the user will have to click on the ‘See Stats’ button again. The only test we can perform for this method is to visually witness the above interactions.

2.2) Integration

a. Event-based architecture (controller)

I. This test will be designed to test the '*Click Play Button'* action in our state diagram for our controller. In order to begin this state, it must be our user's turn, and the user must not have clicked anything, or have clicked the screen after clicking a button. To begin this test, we should assert that the controller's state is 'READY'. Following this, we should click on the button that says 'Play card'. After this button, we must assert that the controller's state is 'PLAY'.

II. This test will be designed to test the '*Click Tell Info Button*' action in our state diagram for our controller. In order to begin this state, it must be our user's turn, and the user must not have clicked anything, or have clicked the screen after clicking a button. To begin this test, we should assert that the controller's state is 'READY'. Following this, we should click on the button that says 'Tell Info'. After this button, we must assert that the controller's state is 'READY'.

III. This test will be designed to test the '*Click Discard Button*' action in our state diagram for our controller. In order to begin this state, it must be our user's turn, and the user must not have clicked anything, or have clicked the screen after clicking a button. To begin this test, we should assert that the controller's state is 'READY'. Following this, we should click on the button that

says ‘Tell Info’. After this button, we must assert that the controller’s state is ‘DISCARD’.

IV. This test will be designed to test the ‘*Click AI Button*’ action in our state diagram for our controller. In order to begin this state, it must be our user’s turn. To begin this test, we should assert that the controller’s state is now ‘AI.’

V. This test will be designed to test the ‘*Click Display Discard*’ action on our event-based controller architecture. In order to begin this test, the discard pile must not be visible on the screen, and the controller must be in the ‘READY’ state. We must first assert that we are in the ‘READY’ state, and after we click on the button that says ‘Show Discard Pile’, we must assert that we’re still in the ready space. Additionally, we must visibly confirm that the discard pile is able to be viewed on screen.

VI. This test will be designed to test the ‘*Click Show Stats*’ action on our event-based controller architecture. In order to begin this test, the stats must not already be visible on the screen and the controller must be in the ‘READY’ state. First, we will assert that we are in the ‘READY’ state, and after we click on the ‘Show Stats’ button, we will assert we are still in the ready space, and we will make sure that the stats are able to be viewed on the screen.

VII. This test will be designed to test the ‘*Click Target Card*’ action on our event-based

architecture for our controller. This test will begin by asserting that the controller's state is 'TELL'. Following this, the human user will click on one of their teammate's cards. After this click is performed, we must assert that the controller is in the 'INFORMATION' state.

VIII. This test will be designed to test the '*Click Board*' action from any state except the INFORMATION state. A test must be done for each of the following states: 'PLAY', 'READY', 'DISCARD', 'TELL'. Assuming that variable X is one of these states, we must first assert that the controller is in state X. Following this assert, we will require the user to click on the board, at any location that is not one of their cards. Following this click, we must assert that the controller is in state READY.

IX. This test is will be designed to test the '*Click Board*' action from the 'INFORMATION' state. At the beginning of this test, we must assert that the controller is in the state 'INFORMATION'. Following this, the human user must click on the board, at any location that is not one of his team mates card. After completing this action, we must assert that the controller is in the state 'TELL'.

X. This test will be designed to test the '*Click Card to Discard*' action. First, the test will assert that we are in the 'DISCARD' state. Then, following clicking the card to be discarded, the test will

assert if we are now in the ‘WAIT’ state. Lastly, it will check to make sure by clicking the desired card, that the proper JSON message will be sent out.

XI. This test will be designed to test the ‘*Click Target Card to Play*’ action. First, the test will assert that we are in the ‘PLAY’ state. Then, following clicking the card to be played, the test will assert if we are now in the ‘WAIT’ state. Lastly, it will check to make sure by clicking the desired card, that the proper JSON message will be sent out.

XII. This test will be designed to test the ‘*Click Color/Number*’ action on our Event-based controller architecture. At the beginning of this state, we must assert that the controller is in the ‘INFORMATION’ state. The user will then click on one of the two buttons that are on the screen: The button that says ‘Number’ or the button that says ‘Color’. Two tests must be performed at this point. The first test will begin with the user clicking on the button that says ‘Number’. Following this action, we must assert that a JSON message is generated that includes the player we’re telling information to and what information we’re telling them. If the user clicked the ‘Color’ button, we must assert that a JSON message is generated that includes the player we’re telling information to and the color that we’re telling them about. Following both of these

cases, we must assert that the controller is in the READY state.

XIII. This test will be designed to test the '*Receive Turn Notice*' action. First, the test will assert that we are in the 'WAIT' state. Then we will assert that the message is informing us that it is our turn. Then, following receiving the message, we will assert that we are now in the 'READY' state.

b. <u>Data-flow architecture (AI)</u>

I. This test will be designed to test the "*Get Game Start*" process in our data flow diagram. Firstly, the user will receive a JSON message from the server, and it will state whose turn it is. We must assert that the JSON message contains information containing that it is our clients turn (IE, the id of the turn must match the id stored in our board). Next, the user are able to get information from the game board. We will assert that the information supplied to this area matches the information stored in the bored (including all players' hands, the discard pile, and the center pile).

II. This test will be designed to test the "*Tell Information*" process in our data flow diagram. Firstly, this test will assert that AI can read the number of tokens left in the game, and that this number is greater than zero. We will then assert that the information about every player's hand visible to the AI, the discard pile, and the center

piles, are equal to the information stored in the board. We will then assert that this process creates a JSON file that contains both a player to tell information to and what kind of information to tell. Finally, we must also assert that the given player has at least one instance of the given information to tell.

III. This test will be designed to test the "*calculate prob card*" process in our data flow diagram. This test must assert that the AI has information about the discard pile, each players' hand, and the center piles. We must assert that this information is equivalent to the information stored in the model. We must then assert that the most probable card available is the card that is calculated. We can do this by creating a circumstance in which the most probable card available is a red one, and that this process creates a Card object that contains information about a card that is a red one.

IV. This test will be designed to test the "*Choose Discard Play*" process data flow diagram for our AI player. After the AI receives card data from the last process, it will determine whether to do the play action or discard action. In order to test this, we must consider two cases. The first is where it is better to play the card provided as input. In this instance, we must first assert that the card is playable. We must then decide to play a card in our hand, and then produce a

JSON message containing information about the position of the card we wish to play in our hand. We must then assert that a card is removed from our hand (whether it is what we predicted or not is irrelevant), and that a new card is added. In the second case, we must assume that we want to discard the card that we have received as input. First, we must assert that there is at least one information token available in the game. Secondly, we must assert that a JSON message is produced that contains information about the card we wish to discard. We must then assert that a card is removed from our hand (whether it is what we predicted or not is irrelevant), and that a new card is added.

V. This test will be designed to test the "*Choose Move*" process in our data flow diagram. First, we must assert that we have two JSON messages. One should contain information about who we want to tell the specific information to, and the other should contain information about if we want to play or discard a card. This test will have a few instances. The first instance is when we can tell information to a player in which their move will be immediately beneficial. If we can tell a player after us that they have a card that is immediately playable, given the center piles, then the process should send the JSON file containing the details about

telling information through. If this is not an option, however, this process should send information about playing or discard a card through.

c. <u>Model-View-Controller Architecture</u>

I. Testing losing a fuse: To begin this test, it must be our player's turn. We will then simulate sending the server a move that is not valid (IE playing a 5 when no cards are played). After this, we must assert that the number of fuses we had decreases by one. We must also check to see that the view redraws itself with one less fuse visible.

II. Increasing the number of tokens (on discard): To begin this test, we will assert that the model is storing less than 7 information tokens. Additionally, we will assert that it is our clients turn. We will simulate discarding a card in our hand. We must assert that this card appears in the discard pile, and that the number of information tokens stored in the bored increases by one.

III. Testing playing a card: Following the test that involves sending a card to the server in a JSON message indicating that a card was played. We must assert that a card has successfully been removed from our hands, and that a new card is drawn.

IV. Testing discarding a card: It must be our clients turn to perform this test. Following the test that involves sending a card to the server as a JSON message indicating that a card was discarded. We must assert that the card has successfully been removed from our client's hand and that our client draws a new card. Additionally, we must also check that the discarded card appears in the discard pile, and we must perform test 2 above.

V. Testing receiving information from another player: Upon receiving information from another player, our model will be updated. Upon updating, our view must be redrawn. This test will be simulated by having our client first having no information about their cards. The server will then send them a JSON message indicating a type of information. Our model must then update to reflect this, and the view must show information. For instance, if our client has a hand of full red cards (but no given information), they will then receiving information informing them of their red cards. All of their cards must then be drawn as red to show this information. We must also assert that the cards in the hands reflect this change by asserting that the boolean containing the visibility of the color of each card in their hand is turned on.

2.3) Unit

I. Class: *Hand*

1) The first test for this class will test if we can add a card to the hand, and will test the method '*addCard(Card)*'. Given a card, we must simply add said card to the Container of cards stored in the class. To test this, we will create a *Mock(Hand)* object, with a hand containing 4 cards. The cards do not particularly matter. We will then create another card and insert it as an argument to this function. Following this, we must assert that this new card is contained in the container (if the said card was already in the hand, then we will check to see if there is now one more of that card in the hand), and that the length of the container increased by 1 (IE it is now 5).
2) This test will be designed to test the method '*removeCard(Card)*'. In order to conduct this test, we will construct a *Mock(Hand)*, with a *Mock(ArrayList<Card>)* Field containing a red, white, blue, green and yellow one. We will then pass in an instance of a yellow one card. After this function call, we must assert that the *Mock(Hand)* object no longer has the yellow one in it's *ArrayList<Card>* field.
3) This test will be designed to check the CheckHit(float, float)' method of this class. In order to test this, we will first create a *Mock(Card)* instance, with xCorner = 0 and yCorner = 0, and add this to the hand. We will then pass in two float values and check that they

lie within the card. In one instance of this test, the two values will lie within the area of the card, and this method will return true. In the other test, the two values will not lie within the area of the card, and this method will return false.

II. Class: *DiscardPile*

1. This test will be designed to check the '*ReceiveCard(Card)*' method of this class. This method is simply stored discard card to container of cards stored in the class. To test this, we are able to create *Mock(DiscardPile)* object, with a pile containing 3 cards. It does not matter with different or same cards. Then, we could create another card object to insert. Now, the new card is in discard pile, and the number of card in pile is 4.
2. This test will be designed to check the '*getPile()*' method of this class. The purpose of this test is sending all of the cards that have been discarded in a textual style. To test this method, we could simply see how many cards in the discard pile, and what kind of card pile have. Then, simply calling this function to return summarizing all instances of card stored in the field '*Cards'* in string type.

III. Class Server:

1. This test will be designed to test the 'connectPlayer' method of this class. We must create a Mock(Board) instance and attach it to a server. In order to test this method, we must

simply create a Mock(Player) instance, and pass it into this method. We must then check that this player is stored in our instance of the board.

IV. Class: *AI Player*

1. This test will be designed to check the '*calculateMove()*' method of this class. This method is designed to calculate the probability of the most likely color and number in the user's hand. It returns the most likely card in the player's hand. To test this method we will create a mock hand for the AI and other 'players' and start with simple cases. We will start with giving the other players each a hand with the same color with ascending cards. So player 1 would have all reds with ascending values (1, 2, 3, 4, 5), player 2 would have all whites (1, 2, 3, 4, 5), and player 3 would have all greens (1, 2, 3, 4, 5). And the discard pile would have all blues. So the AI's calculation for the probability of what color it has should return a very high probability that it has a yellow card with value one. Since, all the other colors have been seen and ones are the most common value. We will continue to test scenarios and gradually get more complex. For more difficult cases we will have to check if the method is working ourselves, using our own calculations.
2. This test will be designed to test the '*CalculateTellInfo*' method. This method is designed to calculate what kinds of information

the AI could provide to other players. This function takes in an array of type Player as input and returns a JSON message to the '*ChooseMove*' method. To test this method we will create mock hands, like in the '*CalculateMove*' method. First, we will create a mock situation where there are no cards played. The AI should then check for the lowest numbered cards in each person's hands and choose the player with the most lowest cards. We will then assert that the method returned a JSON message indicating that they chose that player and the information they will provide is correct. We will then create mocks with slightly more complex situations (ex; cards are played on the board), and again assert the output.

3. This test will be designed to test the '*ChooseDiscardPlay*' method. This method is designed to take in the most probable card from the '*calculateMove*' method and compare the benefit of playing that card, versus discarding that card. It will then return a JSON message indicating which is more beneficial to the '*chooseMove*' method. We will then, again, create a mock situation with mock players and mock hands. We will check the benefit of the two possible moves, based on the situation. In order to test this, we will create a mock instance of a hanabi game in which each color's played pile has up to the numerical value 2 played in it. We

will then pass in two JSON messages to this method: One containing information about playing a green 4, and another about telling a team member about a green 3. This method should return the JSON message about telling a team member about the green 3. We can assert this. If we conduct a similar test, and pass in JSON message about playing a green 3 or telling a team member about a green 4, this method should return playing the green 3.

4. In order to test the '*chooseMove*' method of this class, we will first create two Mock(JSON) objects. One will contain information about playing a green one card, while the other will play information about telling player two about a green five. We will generate a Mock(Board) instance, with players one and two, and a center pile containing nothing. We will then assert that this method is going to return the JSON message that contains information about playing a green one.

V. Class: *Human Player*

1. This test will be designed to check the method '*joinGame(int)*' of this class. To test this class we will create a mock server with id of five and we will check if the server can add players in it.

VI. Class: *GameManager*

1. This test will be designed to check the "*createGame(float,int,int)*" method of this class. This method is designed for creating a game

with three parameters. The float is the ID gamemaster wants to create, the second is numbers of players, and the third one is length of time each player will be allotted per turn. To test this method we will create a mock "*createGame*" with three parameters. We can test the ID that other game has; it should return "message show unavailable," and we can test if the player number is less than two and over five; it should return a message of "invalid number." We can test the length of time as a negative number, which it should show "invalid length."

VII. Class: Board

1. This test will be designed to test the method '*NotifySubscribers()*'. We will create a *Mock(View)* instance, and make this instance a subscriber of the board. We will then call this function, and we must assert that the *BoardChanged()* function in the mock view is reached.
2. The second test this class must have will be to check if we can properly detect clicking on a card that is stored in a players hand in the model. To do this, we will test the method '*checkHit(float, float)*'. To do this, we will generate a *Mock(Player)* instance, with a *Mock(Card)* instance of a card that contains xCorner and yCorner values of 0. We will then create a mock(Player) instance and insert this mock card into the hand. We will then add this

player to an instance of a *Mock(Board)* object. Following this, we will perform two tests. The first will be done as follows: Call *CheckHit*(10, 10), and assert that it's return value is true. Following this test, we will then call *CheckHit*(100, 100), and assert that it's return value is false.

3. This test will be designed to test the '*reactToJSON(JSON)*' method of this class. In order to conduct this test, we will construct a *Mock(Board)* instance with one player that has an arbitrary group of cards. We will give this *Mock(Player)* object an ID of 1, and we will give our board an id of 1. Two tests must be conducted to deduce this functions functionality. The first is when the JSON message indicates our turn. For this test, we will pass in a JSON message that indicates that it is our turn to the *reactToJSON* method. We will then assert that we properly begin our player's turn. Following this, we will then pass this method a JSON message that does not indicate our turn (For instance, maybe the JSON message indicates that it is player 4's turn), and assert that we do not begin our player's move.
4. In order to test the '*addSubscriber*' method of this class, we will first create a *Mock(Board)* object. We will then create a *Mock(View)* instance. We will then call the method *addSubscriber* with this view as input, and

following this method call we will assert that our list of subscribers contains this view.

5. In order to test the '*addPlayer'* method of this class, we will first create a *Mock(Board)* object. We will then create a mock instance of a Player object. We will then call the *addPlayer* method with our mock player object as input. We must then assert that our mock board's list of players now contains the player that we added.

VIII. Class: View

1. In order to test the '*onDraw(Canvas*) method of this class. The purpose of this method is to draw the model. To test this method, we are able to create *Mock(View)* and *Mock(Board)* instance, with ID = 2. In the board instance, we will have an instance of a *Mock(Player)*, with id=1, with one *Mock(Card)* that is a red one. We will have a second *Mock(Player)* instance in the board, with id=2, with one *Mock(Card)* in their hand with a value of a blue one. We will then call the *onDraw* method of this class, and we will visually assert that one black card is drawn at the bottom of our screen, while one red card with the number one on it is drawn in the top left corner of the screen.

IX. Class: Controller

1. In order to test the '*onClick(Event)'* of the controller, we must first create an instance of a *Mock(Controller)* in the READY state, and a *Mock(Board)*. We will also create 5 *Mock(Card)*

instances. The colors and numerical values of these cards will be arbitrary, but their xCorner and yCorner values will be as follows: (0, 0), (30, 30), (60, 60), (90, 90), (120, 120), respective to (xCorner, yCorner) for each card. Following this, we will create a *Mock(Hand)* instance and add these 5 cards to the hand. We will then create a *Mock(Player)* instance and add this hand to the player. We will then add this player to the board instance. Following this, we will do the following tests: Detect a mouse hit on the play button, assert that the controller's state is now "PLAY". We will then simulate a click with x and y values of 500 each. We will then assert that the controller's state is "READY". We will then simulate a click on the "Play" button and then a click with x and y values of (10, 10). We will then assert that the first mock card that we created is no longer in the players hand, and that our controller's state is 'WAIT', and we will manually change the controllers state back to 'READY'. We will then simulate a click on the "Discard" button, and assert that the controller's state is 'DISCARD'. We will then simulate a click with x and y values of (40, 40), and assert that the controllers state is 'WAIT'. Following this, we will manually change the controller's state to 'READY'. We will then assert that the second mock instance card is no longer in our mock player's hand. Following this, we will generate a

click event on the “Tell Information” button, and assert that the controller’s state is in the ‘TELL’ state. We will then generate a hit on card three by passing in an event with coordinates (70 , 70), and assert that our controller is in state ‘INFORMATION’. We will then simulate a click on the button that says “Color”, and assert that our controller is in the ‘WAIT’ state.

3) Alternate Tests

3.1) <u>End-to-end</u>

I. ‘*Create Game*’ use case, step 1: In this instance, we will assume that the user attempting to create a game already has a game created. Inputting the given bash command line, with any settings, will indicate to the player on the command line that they have a game created already. They will then be asked if they want to force-quit their game. They will then need to input ‘y’ or ‘n’.
   If the user inputs ‘y’, we must check to see that a new game has been created, and that they’re not connected to their old game.
   If the user inputs ‘n’, we must check to see that they’re still connected to their old game.

II. ‘*Join Game*’ use case, step 1: In this case, if the room they are trying to join doesn’t exist we will indicate to the user that the game was not a valid game to join.

III. ‘*Human Play Card*’ use case, step 2: This step

will require several things. The first will be that it is currently the users turn. The second is that the user must have clicked on the button that says “Play Card”, but nothing else. If these requirements are met, the screen should indicate to the player to “Select a card”. The player will then click on anything other than their own card. After clicking, the screen should indicate that the player must select a move. We also check that the player’s hand did not change, and that it is still their turn.

IV. ‘*Human Discard Card*’ use case: For this action to occur it must be the user’s turn and the team must have less than seven information tokens. If it is currently not the player’s turn, then clicking on the discard button will do nothing. If the team does not have less than seven information tokens then, following clicking the discard button, the client should indicate that the player already has seven tokens and cannot discard. Then the player will be prompted to select a new move.

3.2) Integration

a) Event-based Architecture (Controller)

I. This test will be designed to test the ‘*Click Discard Button*’ action in our state diagram for our controller. Specifically it will handle the case for when the team has seven information tokens. In order to begin this state, it must be our user’s

turn, and the user must not have clicked anything, or have clicked the screen after clicking a button. We should also assert that the controller's state is 'READY'. Following this, we should click on the button that says "Discard Card". After clicking the button, the client should inform the player that this is not a valid action because they do not have less than seven information tokens.

b) <u>Data-flow architecture (AI)</u>

I. This test will test will be designed to test the AI's '*Choose Discard Play*' process from '*Calculate prob Card*'. After the AI receives card data from last process, it will determine whether to perform the play action or discard action. This alternate test will make sure the AI cannot perform a discard action when there are not less than seven information tokens in the game. Instead, the AI will choose the play action and send the proper JSON message.

II. This test will be designed to test the '*Tell Information*' process in our data flow diagram. This test will make sure the AI cannot give information when there are zero information tokens. If there are zero information tokens then the client will send a JSON containing an empty string.

III. This test will be designed to test the AI's **'***Choose Move***'** process. The test will assert that the inputs are two JSON files. One containing an

empty string and the other containing the information about playing or discarding a card. In this case, we must assert that this process returns the JSON string about playing or discarding the card.

3.3) <u>Unit</u>

I. For the *'CalculateTellInfo'* method in the AI player class, an alternative test will be when the team has no information tokens. In this case we will assert that the function is returning an empty JSON message.We will do this by making a mock(AI) instance and a mock(Board) instance. In this circumstance, this function should return an empty JSON message.

II. In the '*onClick()*' method of the controller class when we click on the discard button and our team has seven information tokens, we need to be able to notify the user that this action is not possible. In order to test this, we must have a controller that is in the 'READY' state, and have a model that has seven information tokens. We must then simulate a click on the discard button. We must assert that, after this click, the controller's state is still 'READY', and visually confirm that a notice appears indicating that discarding is not currently a valid option.

III. In the controller class, we must run an alternate test on the '*onClick()*' method. This test will aim to verify that a player can not attempt to give information when they have 0 information tokens. In order to test this, we will create an instance of a

Mock(Board) object with 0 information tokens, and attach a controller to it. The controller's state must be the 'READY' state. We will then simulate a click on the "Give Information" button. Following this simulated click, we must check that the controller's state is still 'READY', and visually verify that a message on screen has appeared indicating that giving information is not currently an option.