SOFTWARE TESTING PLAN

for

Game Client for Hanabi Card Game

Version 1.0

Prepared by Alexander Lavis, Caim Chen, Justin Pointer, Noah Kovacs, Sajith Rajapaksa

Group E3,CMPT 370, Department of Computer Science - University of Saskatchewan

March 14, 2019

Contents

1 Introduction

The purpose of this document is to outline and describe the details for the project's testing. This is its test plan. It entails three major testing categories. First, it will discuss all of the relevant End-to-End tests, which will cover how all modules work together. Then, it will also discuss each of their subsets, which are Integration Tests. Each Integration Test will test the full functionality of each module. And finally, each Integration Test will cover its subsets, Unit Tests. Unit Tests test and cover specific methods, and compare them against given and expected results. Since no implementation has been made at this time, these Unit Tests only make use of Black Box Testing.

It is also important to Note that this document makes frequent references to the Requirements and Design Document. For more information on the use-cases being discussed, see the Requirements document.

2 End to End Testing

Now that the Introduction is complete, the End-to-End tests will now be outlined. In total, there are 5 End-to-End tests. They are as follows:

- Creation To Waiting
- Joining To Waiting
- AI Controlled Play
- Human Controlled Play
- Full Game

Listed below are more in-depth explanations for each of the End-to-End Tests. The first test described is Creation to Waiting.

2.1 Creation To Waiting

In this section, the testing methodology is presented for creating a Hanabi game by the Host player. This end-to-end test includes two integration tests. It tests the whole process of creating a game, then the transition to the waiting state. This includes starting the game, opening the creation menu, communicating with the server, transferring to the waiting state, and finally initialization. The purpose of this test is to ensure that the creation of the game will perform as a connected system.

2.2 Joining To Waiting

In the Joining to Waiting End-to-End Test, the testing methodology is presented for creating a Hanabi game by the Host player. This End-to-End Test includes two Integration Tests. It tests the whole process of joining a game. This includes starting the game, opening the join menu, communicating with the server, transferring to the waiting state and initialization. The purpose of this test is to ensure that the joining of the game will perform as a connected system. Although this is similar to the creation End-to-End Test, it differs in the required functionality of the menu used and the JSON messages sent to the server.

2.3 AI controlled play

The AI controlled play End-to-End Test checks if the AI controlled players in an active game can perform the required range of possible actions. This End-to-End Test requires three integration tests regarding the AI's actions: attempting to play one of their cards, discard one of them, or give a hint to another player (AI or Human).

This test ensures that when a new game is started (that includes at least a single AI controlled player and it is an AI player's turn to make a move), that it can play a card onto the card stack. On a different turn, the AI player can also discard a card from its hand, and again, on a different turn, an AI player can give another player a hint as to what's in their hand.

2.4 Human Controlled Play

The Human Controlled Play End-to-End Test checks if the Human players in an active game can perform the required range of possible actions. This End-to-End Test requires three Integration Tests. First, it checks when game reaches a human player's turn. Second, it tests if the human player can select a card to play. Once it is selected, they can then place that card onto either the fireworks stack or the discard pile (because of duplication). If it is placed into the discard, the info token count should be changed accordingly. Finally, it tests if the human player can select a card in other player's hand and inform them of a number or suit by consuming one info token. Info token number should change accordingly.

This test ensures that when a new game is started (and it is a Human player's turn), it can play a card onto the card stacks. Then on the succeeding turn, a Human player can discard a card from its hand, and, again on a different turn, a Human player can give another player a hint as to what's in their hand.

2.5 Full Game

In the Full game End-To-End Test, it will check as to whether the game is able to be completed/ able to reach the end without error. This occurs in many runs of the game, as there exists many game ending integration that cannot be tested in the same trial. In each trial, it will make use of all integration tests, excluding the aforementioned game ending integration's; only one of these will be used per run. Since there are six game ending conditions, the game must be ran through six times. The game ending conditions include Fuse Burn Game End, Deck Empty Game End, Invalid Action Game End, Quit Game End (detailed in button), Time Out Game End and Teammate Quit Game End.

3 Integration Testing

Now that the End-to-End Tests are complete, the Integration Tests will now be introduced. Integration tests are a combination of smaller unit tests. In total, there are 16 integration tests. They are as follows:

- Create Game
- Join
- Initialize Game
- Button
- Human Play Action
- Human Discard Action
- Human Hint Action
- AI Play Action
- AI Discard Action
- AI Hint Action
- Messaging Connections and Filter
- Fuse Burn Game End
- Deck Empty Game End
- Invalid Action Game End
- Time Out Game End
- Teammate Has Quit Game End

Listed below are more in-depth explanations for each of the Integration Tests. The first test described is Create Game.

3.1 Create Game

The Create Game integration test ensures that the controller follows the steps of Use Case 1: Create Game, which starts the Game State machine then loads the start menu with the create game option. This test checks that a well formed JSON message is sent to the server, and that a game is to be created with a host, the inputted number of human players, and the number of AI players. It then sets the host player into the waiting state of the Game State Controller. Unit tests for this integration test can be found in section 4.1.

3.2 Join

The Join integration test ensures that the controller follows the steps of Use Case 2, Join Game, along with various functions required to construct the game model. It checks that a well formed JSON message is sent to the server, indicating that a player (both human or AI) who joins a created game receives a reply from a test mock (or provided game server).The model is then updated for model consistency, including: Properly working join menus function for human players and the AI joiner functions correctly, Setting each player's is Human Flag correctly and informing all current players that they have joined using JSON messages finally the game models are assembled for all players when game is full. This test involves 9 unit tests, they can be found under section 4.2

3.3 Initialize Game

The Initialization Integration Test ensures that the model is properly constructed for all players, in addition to making the card objects and filling the deck along with various functions required to construct the game model. After receiving the 'game is full message,' from the server, it creates hands from its supplied hand lists. This test has no use-cases, but is required for all other classes to function. This test involves 5 unit tests, they can be found under section 4.3.

3.4 Button

The Button test ensures that the controller follows the steps of Use Cases 6 and 7 the Swap and Quit cases respectively. It checks that the Swap and Quit buttons along with other various buttons used in the program's respective Action Listeners are functioning correctly. This test involves 7 unit tests, they can be found under section 4.4

3.5 Human Play Action

The Human Play Action Integration Test ensures that the controller follows the steps of Use Case 3: Play Card, from the Requirements Document section 2.3.3. In this case It checks the human Input variation of the use-case separate from later AI tests. Takes in Human input for play action. Then checks that a well formed JSON message is sent to the server telling that the Play card action received a reply from the actual card played. The model is correctly updated for model consistency, including: Playing the card to the appropriate stack if valid or goes to discard pile if not valid and decrements fuses. Then draw a new card for player and place in their hand and inform all players of new card. Unit tests can be found under section 4.8.

3.6 Human Discard Action

The Human Discard Action Integration Test ensures that the controller follows the steps of Use Case 4: Discard a Card, from Requirements Document section 2.3.4. In this case it checks the human input variation of the use-case separate from later AI tests. It takes in human input for the discard action. It then checks that a well formed JSON message is sent to the server telling that Discard Action occurring receives a reply with the actual card discarded. The model is correctly updated for model consistency, including: placing card discarded into Discard Pile, increments available info tokens up by one and drawing a new card for player and place in their hand and inform all other players of the cards value. Unit tests can be found under section 4.8.

3.7 Human Hint Action

The Human Hint Action Integration Test ensures that the controller follows the steps of Use Case 5: Hint from requirements document section 2.3.5. In this case it checks the human input variation of the use-case, separate from later AI tests. It checks that a well formed JSON message is sent to the server telling of the Hint Action being taken and what information they are giving suits or a color receives a reply with the actual cards corresponding to options input. The model is correctly updated for model consistency, including: Change the other players view of hand updated with new information and decrements available information tokens by one. Unit tests can be found under section 4.8.

3.8 AI Play Action

The AI Play Action integration test ensures that the controller follows the steps of Use Case 3: Play Card from requirements document section 2.3.3. In this case It checks the AI's ability to function during the Play Action test. The tests involved check the AI table of responses calculated from current model state then selects best card. It checks that a well formed JSON message is sent to the server telling of the Play card action receives a reply with the actual card played, and that the model correctly updates for model consistency including: Playing the card to the appropriate stack if valid or goes to discard pile if not valid and decrements fuses. Then draw a new card for player and place in their hand and inform all players of new card. Unit tests can be found under section 4.7.

3.9 AI Discard Action

The AI Discard Action integration test ensures that the controller follows the steps of Use Case 4: Discard a Card from requirements document section 2.3.4. In this case It checks the AI's ability to function during the Discard Action test. The tests involved check the AI table of responses calculated from current model state then selects best card. It checks that a well formed JSON message is sent to the server telling of the Discard Action being taken receives a reply with the actual card, and that the model correctly updates for model consistency Including: placing card discarded into Discard Pile, increments available info tokens up by 1 and drawing a new card for player and place in their hand and inform all other players of the cards value. Unit tests can be found under section 4.7.

3.10 AI Hint Action

The AI Hint Action integration test ensures that the controller follows the steps of Use Case 5: Hint from requirements document section 2.3.5. In this case It checks the AI's ability to function during Hint Action test. The tests involved check the AI table of responses calculated from current model state then selects best card. It checks that a well formed JSON message is sent to the server telling of Hint Action being taken and informs the server with a JSON message corresponding to information selected to inform about for player selected, and then the model correctly updates for model consistency including: Change the other players view of hand updated with new information and decrements available information tokens by one. Unit tests can be found under section 4.7.

3.11 Notification

The Notification integration test ensures that any notice received by the server is properly displayed for the player.This test combine multiple unit tests from the messenger class to receive and parse the JSON, filter to call correct menu to be used as display and menu click actions. Unit tests can be found under section 4.9.

3.12 Fuse Burn Game End

The Fuse Burn Game End Integration Test ensures that the game ends on the condition of all the fuse tokens being exhausted. This integration test makes use of 2 number of unit tests can be found under section 4.10.

3.13 Deck Empty Game End

The Deck Empty Game End Integration Test ensures that the game ends on the condition that there exists no cards in the deck. This integration test makes use of 2 of unit tests as listed in section 4.11.Note that this game ending behavior mimics the game ending behaviour of Quit game button (outlined in the button integration test).

3.14 Invalid Action Game End

The Invalid Action Game End Integration Test Ensures that the game ends on the condition that an invalid card is being played (for example, a Red 6). If this is the case, 1 additional run through the action decider will conclude the game if it is invalid once again. Note that this game ending behavior mimics the game ending behaviour of Quit game button (outlined in the button integration test). This integration test utilizes the unit tests listed under section 4.12.

3.15 Time Out Game End

The Time Out Game End Integration Test Ensures that the game ends on the condition that a player is idle past the allotted turn time. Note that once this is triggered, its behavior mimics the game ending behaviour of Quit game button (outlined in the button integration test). It thus makes use of the Unit test cases listed under section 4.13.

3.16 Teammate Has Quit Game End

The Teammate Has Quit Game End Integration Test ensures that the game ends on the condition that a human players has left the game. If this trigger has happened, note that the game ending behavior mimics the game ending behaviour of Quit game button (outlined in the button integration test). This integration test makes use of the Unit test Cases listed under section 4.14.

4 Unit Testing

As described in the integration tests, this document will move on to defining the unit tests that comprise each of the previously described integration tests. Here, the document details the design of the bulk of the testing to be done by our plan.

4.1 Create Game

4.1.1 Create State Change

Once the player clicks CREATE button on the start menu. This unit test checks if the current state variable of the Game state machine class is set correctly to Create state. Purpose of this test is to make sure the clicked listener worked adequately to start the Game State Machine.

4.1.2 Create Menu Display

Visually checks if the create menu is displayed correctly. Ensure all the fields mentioned in requirements are met including NSID, Number of Human players, Number of AI players and time limit.

4.1.3 Information Intake

Once Create button is clicked, check all the required information is collected from the input by making sure that the model sets all the variables mentioned previously then checking all of the variables present values in the host's model.

4.1.4 JSON translation

Given the needed information to the Messenger ensures the correct JSON string is created. This is to ensure that the communication between the server and messenger can be made. This is done by creating a simple Test mock that sends a string message into the function. Then receives the output JSON string and compares to expected output.

4.1.5 Send JSON to server

Given a message in the create Jason format, messenger successfully sends the JSON file to the server. This can be tested by waiting for a reply from the server.

4.1.6 Receive JSON:

Checks the messenger correctly received a JSON message counting create related information after sending a create JSON to the server.

4.1.7 Parse JSON

Given A JSON file in with the create the mock tests whether the input was parsed correctly into a list containing the expected information.

4.1.8 Filter Message

Given a list of string containing game creates information, the filter should notify the host with the required information.

4.1.9 Waiting State change

Once a game successfully created, the game state machine should change the current state to waiting state. In this unit test, we test this occurs by creating a game and checking the current state variable of the game state machine function.

4.2 Join

4.2.1 Join State Change

Once the player clicks JOIN button on the start menu. This unit test checks if the current state variable of the Game state machine class is set correctly to Join state. Purpose of this test is to make sure the clicked listener worked properly to start the Game State Machine.

4.2.2 Join Menu Display

Visually checks if the join menu is displayed correctly. Ensure all the fields mentioned in requirements are met including NSID, game ID, and secret.

4.2.3 Information Intake

Once Join button is clicked, check all the required information is collected from the input by making sure that the model sets all the variables mentioned previously then checking all of the variables present values in the payer's model.

4.2.4 JSON translation

Given the needed information to the Messenger ensures the correct JSON string is created. This is to ensure that the communication between the server and messenger can be made. This is done by creating a simple Test mock that sends a string message into the function. Then receives the output JSON string and compares to expected output.

4.2.5 Send JSON to server

Given a message in the join JSON format, messenger successfully sends the JSON file to the server. This can be tested by waiting for a reply from the server.

4.2.6 Receive JSON:

Checks the messenger correctly received a JSON message containing join-related information after sending a join JSON to the server.

4.2.7 Parse JSON

Given A JSON file in with the join mock tests whether the input was parsed correctly into a list containing the expected information.

4.2.8 Filter Message

Given a list of string containing game join information received from the server, the filter should notify the player with the required information.

4.2.9 Waiting State change

Once successfully Joined, the game state machine should change the current state to waiting state. In this unit test, we test this occurs by joining a game and checking the current state variable of the game state machine function.

4.3 Initialization Action

4.3.1 Card

Theses test confirm that cards are correctly made. This is vital for creating a deck and dealing a hand. Without the correct cards or the correct format, the game cannot be played successfully. Most of these tests are get/set tests.

4.3.2 Create Card

Since cards are correctly formatted, they can now be checked for the right values, in addition to being placed in the deck. If any card is missing or incorrect, then the game will not function correctly. Check that the list contains all the necessary cards. This involves iterating through the card list, and checking that each entry exists in a fixed array of all the necessary cards. If one does not exist, or one is missing, the test fails.

4.3.3 Fill Deck

Check that the deck contains the same cards from the list output from createCard, or, check it against the array of fixed cards. Check that the total number of cards in the deck is equal to the total number of cards available.

4.3.4 Deal Hand

If the hands are not correctly created, or if the wrong number cards end up in hand, then they cannot be played correctly. This test ensures the correct creation of a hand. After creating a hand check that the players' hands do not have empty values and that the deck is missing the number of distributed cards.

4.3.5 AI Join

The purpose of this test is to see if the AI successfully joins a room of players. If so, then the game coexists with a set of humans and AI and is ready to be played. To test this call the AI Join once and after check that the state of the AI is TRUE, a new model was created and that the player waiting number decreased.

4.4 Button

4.4.1 Swap Button

Check to ensure that clicking on the button registers. Mainly, this will check that the button does have an action listener, and therefore works as intended. A Boolean check can be made against the button being clicked. Once the user clicks on the button, are variable should check if it was clicked. A part of this validation can only be made by testing Button Clicked. Note that it is possible that the player has already swapped. If this is true, the integration testing will be identical, in that the human input can be made to press the swap button, and the AI will be switched with its human counterpart at the end.

4.4.2 Button clicked

Use a Boolean to check if the button clicked value is true. If not, the test fails. This test lets us know that the buttons to call an action work..

4.4.3 Swap Button Listener

Now that the Button is clicked, the button needs behaviour. This behaviour will create a menu to appear, prompting the user for more input. Check that a connection to the swap menu is made. This will be done via visual confirmation. If a menu does not appear, then the connection was not complete.

4.4.4 Button Menu

The purpose of this test is to ensure proper behaviour on the button, and to ensure the menu buttons give proper information.here are two buttons on this menu. These follow the same steps as the swap button but has different behaviour defined under the action listener. First, Check the “NO” Case via user input. This indicates the player wishes to retract the action. Return values of the functions indicated in the leading unit tests can be used to compare its success. For example, ButtonClicked should return true if it was clicked. Following this, the menu should disappear. This can be tested by checking a variable for if the menu is active. Also, by visual confirmation, the menu will no longer be present. The game will proceed as normal if the test was successful, and the integration path ends. If the selected button is “YES,” then the following must be done to test. First, it must pass all basic button test, outlined earlier. Second, it must connect to the model. The model will then call a set of functions for swapping the player with an AI. To test this, the test should check the functions return values on success cases. If the return values are not the expected value, then the test will fail.

4.4.5 Quit Button

A Boolean check can be made against the button being clicked. Once the user clicks on the button, are variable should check if it was clicked. A part of this validation can only be made by testing Button Clicked.

4.4.6 Quit Button Listener

After the Button is clicked, the button needs behaviour. This behaviour will create a menu to appear, prompting the user for more input. Check that a connection to the quit menu is made. This will be done via visual confirmation. If a menu does not appear, then the connection was not complete.

4.4.7 Alternative Quit Button

The game has not yet started, and the player presses quit, then a series of messenger tests will be made to remove the player immediately. This can be confirmed visually. See messenger test cases, as well as game ending tests.

4.5 Model

4.5.1 Model Consistency Framework

After the Model of the game has been changed using a setter function, a Model Consistency Test (MCT) must be done. Moreover, it involves using getter function(s) before and after the use of its respective setter function(s) and then comparing the information ensuring that the model has been changed.

Generics:

getModelAttribute(): data item
setModelAttribute(data item): void

The Generics represent the general setup of all the getter and setter methods used in the program design. As such all such methods can be tested using a template mock test that inputs the data item expected by the setter and then returns it using the getter this allows for quick and easy automatic testing of all getter's and setters throughout the program.

4.6 Actions

Testing Pre-condition:
After Human or AI players decide the action they want to perform, the action class is called to execute the given action.

Testing purpose:
player's action can be properly executed by the action class. Messenger class need to be informed after execution of an action.

4.6.1 Play Action

Test if play Card function can generate the correct format of JSON message for play card action, and successfully parse the message to the messenger. After messenger receives replies from the server, execution of the action will be performed unless the desired action is invalid. If invalid action appears, test whether that player gets prompt helpful notify or not. Otherwise, test if the model gets appropriately updated to keep the model consistency. To be more specific, play a card should invoke update firework stack function to indicate the card has been placed on the stack and players earn points. In the end, draw card function needs to be tested since it draws a card to fulfill the player's hand size as long as the deck is not empty.

4.6.2 Discard Action

Test if discard card function can generate the correct format of JSON message for discard card action, and successfully parse the message to the messenger. After messenger receives replies from the server, the execution of the action will be performed unless the desired action is invalid. If invalid action appears, test whether that player gets prompt helpful notify or not. Otherwise, if the model gets appropriately updated to keep the model consistency, to be more specific, discard a card should invoke remove the card to discard pile function and players earn an info token. In the end, drawCard function needs to be tested since it draws a card to fulfill the player's hand size as long as the deck is not empty.

4.6.3 Inform Action

Test if inform card function can generate the correct format of JSON message for inform card action, and successfully parse the message to the messenger. After messenger receives replies from the server, the execution of the action will be performed unless the desired action is invalid. If invalid action appears, test whether that player gets prompt helpful notify or not. Otherwise, test if the model gets appropriately updated to keep the model consistency. To be more specific, inform a card should invoke update model function so the player who receives hint should obtain some card information. As a result of giving a hint, the decrements of info token numbers also need to be tested.

4.6.4 Perform Action

Tests if the Perform function calls the functions playCard, discardCard, and informPlayer function accurately based on its inputs.

4.7 AI Action Decider

Testing Pre-condition:After turn state machine invoke player's turn. The CheckIsHuman function is triggered to verify the current turn player's identity is AI.

Testing purpose: Test if AI player's action can be properly obtained by the AI_Action_Decider class, then parse to the action class.

Unit testing:

- RankAction(ValidActions : ArrayList$< Action >$) : Triplet$< SeatIDInt, CardIndexInt, Action >$)
- QueryTable(Table: Hashtable$< String, Int >$): List$< Pair < String, Int >>$
- CheckBestHint(Hints:List$< String >$): Triplet$< SeatIDInt, CardIndexInt, Action >$)
- CheckValidation(): ArrayList$< Action >$

The AI, Action Decider unit test, queries the updated AI table to produce a list of possible actions by validation. The list of valid actions is then ranked; the output of this ranking process is the action that will be passed onto the action class.

4.8 Human Action Decider

Testing Pre-condition:
After turn state machine invoke player's turn. The CheckIsHuman function is triggered to verify the current turn player's identity.

Testing purpose:
Test if player's action input can be properly obtained by the InGameMenu class, then parse to the action class.

4.8.1 Human Play Action Decider

Play click unit test confirms that steps of section 2.3.3. Player click the desired card for play which has button listener associated with. The selected card information will be parsed to the action class for execution. In particular, the output type of the play click function needs to be a triple which can be accepted by action class as human input.

4.8.2 Human Discard Action Decider

Discard click unit test confirms that steps of section 2.3.4. Player click the desired card for discard which has button listener associated with. The selected card information will be parsed to the action class for execution. In particular, the output type of the discard click function needs to be a triple which can be accepted by action class as human input.

4.8.3 Human Inform Action Decider

Inform click unit test confirms that steps of section 2.3.5. Inform click the desired card for discard which has button listener associated with. The selected card information will be parsed to the action class for execution. In particular, the output type of the play click function needs to be a triple which can be accepted by action class as human input.

4.9 Notification

4.9.1 Notification received

This unit test will tests messenger receiving a notification from the server. To test this an action should be taken which forces the server to send a notification.

4.9.2 Filter Notification

Given a parsed notification message to the filter, it should call the notification menu. To test this functionality using a mock notification to observe the filter.

4.9.3 Notification Menu

Visually inspect the Notification Menu, and ensure that contains the message and a close button.

4.10 Fuse Burn Game End

Test the termination of the Game State under the condition that the last available fuse was burnt. After all the fuses created at the game's initialization, the game should transition away from the Game State and to End Game state.

This should present to the Human players of the game with the end game screen. Visually for inspect the end game screen for the final score and other graphical elements.

4.11 Deck Empty Game End

4.11.1 Last round Flag set

Once all of the cards on the deck is used the model should set a flag for the last round. This can be tested by setting the deck on the model to an empty list and checking the last round flag.

4.11.2 State Change on Last player

Once a full round has been played after last round flag is set the game state should change to end game state. To test this functionality play a full round after the last round flag is set. Check that the state changes, and visually confirm the changes to the end game screen.

4.12 Invalid Action Game End

4.12.1 Notification Received

Once an Invalid notification is received the invalid move flag should be turned to true. To check this, force the server to send an invalid move notification by playing a wrong move and ensure that the notification is received correctly.

4.12.2 Check Invalid Flag

Check the invalid move flag to see if it has been changed to True.

4.12.3 State Change

Once two invalid moves occur, the game is forced to terminate. To check this functionality force the server to send a second invalid move notification and observe is the state change occurs.

4.13 Time Out Game End

4.13.1 Notification Received

Once a terminate notification is received the current game state should be turned to end game state. This will be tested using a mock second timeout JSON on the filter and check the current game state.

4.13.2 Check Time Out Flag

Test to see if the Time Out flag has been raised or has remained lowered.

4.13.3 State Change

Visually check that the final game screen is displaying. This view should include the final score and a close quit button.

4.14 Teammate Has Quit Game End

4.14.1 Notification Received

Once a terminate notification is received the current game state should be turned to end game state. To test this use a mock terminate JSON on the filter and check the current game state.

4.14.2 End Game State

Visually check that the final game screen is displaying. This view should include the final score and a close quit button.

5 Summary

This document presented an outline and description for the test cases needed to be run to ensure the correct functionality of the Hanabi game client. Three types of tests were discussed through the document: End-to-End tests, Interrogation tests and Unit tests. Five End-to-End tests were presented described the intended way of modules working together. Also, sixteen Integration tests were presented describing the overall functionality and behaviour of the Hanabi program. Finally, there were several Unit tests presented not counting the various necessary getters and setters that were too numerous to list but trivial to test.