Software Test Plan

Hanabi Client G3

Authored by:
Joel Siemens <jms676>
Olivier Lacoste-Bouchet <oll096>
Troy Belsher <twb431>
Kole Barber <klb731>
Destin Astrope <dja392>

Introduction

This document serves to outline our test plan for the Hanabi client.

Objectives

The objective of the test plan is to have a set of test cases already planned before coding of any kind. The idea is that we will be coding tests along-side the program so that when the code is finished, it can be tested immediately.

Overview

End to End Tests

Our end to end tests follow each use case. Each end to end tests covers a single use case. Each use case can be found outlined in the Requirements document.

Integration Tests

Our integrations tests are steps within the end to end tests. The tests will ensure that each step, individually, will be covered by a single integration test. This will verify that a small number of classes and functions will operate as outlined in the Design document.

Unit Tests

The unit tests will ensure that individual functions and variables are updating their corresponding fields.

Testing Strategy

End to End Tests

Use Case 1: Host a game (Requirements Document p. 18)

Testing of UI elements on program boot will be mostly visual/manual tests to ensure that all required fields are on the screen. Clicking the rules button should summon the rules screen, and clicking the "close" choice on the rules screen should banish it. Clicking on the "host game" choice should bring up fields to fill in required information to launch a game on the server.

Test 1: A successful attempt to host a game will be when the server has returned the information for other players to join, and enough other players join that the game begins.

Test 2: If there is no connection to the server, the user should be informed of this and the menu should reset to its default state.

Test 3: If input to the text boxes is invalid, then the program should warn the user and prompt them to adjust their inputs.

Test 4: If the server informs the user that a game under the given NSID has already been created, it should pass this message on to the user and and ask permission to force the pre-existing game to close. If so, game should move to waiting for players to join phase. If not, the menu should reset to its default state.

Use Case 2: Join a Game (Requirements Document p. 19)

Tests the different inputs to join a game from the main menu.

Test 1a: Given the appropriate game ID and signature while the lobby isn't full, then the the lobby is joined, user is shown the waiting for player message

Test 1b: Given the appropriate game ID and signature while the lobby is one away from full, then the lobby is joined and the game starts.

Test 1c: Given the appropriate game ID and signature while the lobby is full, then the lobby is not joined and an error message appears.

Test 2: Giving inappropriate game ID and signature to the client, then a lobby is not joined, an error message appears, and the user is returned to the main menu.

Test 3: If there are no active connections, an error message appears and the user is returned to the main menu.

Use Case 3: Reading the Rules (Requirements Document p. 19)

Clicking the rules button should summon the rules screen, and clicking the "close" choice on the rules screen should banish it. The rules will be in a separate window from the main game.

Test 1: Clicking the "rules" button on the main menu should bring up the Hanabi rules in their own window. This will be a visual check.

Use Case 4: Play a card (Requirements Document p. 20)

If this is the first turn of the game the player should receive and correctly interpret a JSON message from the server telling them it is their turn. Otherwise, in later turns the player client should infer it is their turn from the turn order of the game. Controller and interaction model should be set to active state. Play a card action button should initially be clickable, once clicked the player's view should be redrawn with their cards made clickable. All other player action buttons (from section 3.2.1 , use cases 5, 6, and 7) should also be clickable so the player may switch their state to those actions.

Test 1: When any of the active player's cards are clicked, the card should be played. The server will inform the player if the play was successful or not, and all players will be notified as to which card was played. Their client programs will individually determine if the play was successful or not and these should all come to the same conclusion. If successful the correct firework should be added to on all clients, potentially ending the game if it is the final firework to be built. If the play was unsuccessful all clients should remove one fuse token and the card should be added to the discard pile. If any cards are left in the deck, all clients except the active player should be notified of the new card given to the player as a replacement for the played card.

Test 2: This test will be done through visual inspection of the UI elements throughout the use case, ensuring interactability and visibility changes at each point described. Because it is a MVC system, seeing the cards change in the view should confirm that the model was successfully updated as expected.

Use Case 5: Discard a Card (Requirements Document p. 21)

If this is the first turn of the game the player should receive and correctly interpret a JSON message from the server telling them it is their turn. Otherwise, in later turns the player client should infer it is their turn from the turn order of the game. Controller and interaction model

should be set to active state. If there are fewer than eight info tokens, discard a card button should be clickable. Otherwise it should not be interactable. Once clicked it should redraw the players hand with all cards clickable. All other player action buttons (from section 3.2.1, use cases 4, 6, and 7) should also be clickable so the player may switch their state to those actions.

Test 1: When any of the active player's cards are clicked, the card should be discarded. All players will be notified as to which card is discarded. All clients should add one info token to the pool and the card should be added to the discard pile. If any cards are left in the deck, all clients except the active player should be notified of the new card given to the player as a replacement for the discarded card.

Test 2: This test will be done through visual inspection of the UI elements throughout the use case, ensuring interactability and visibility changes at each point described. Because it is a MVC system, seeing the cards change in the view should confirm that the model was successfully updated as expected.

Use Case 6: Give Colour Info (Requirements Document p. 22)

If this is the first turn of the game the player should receive and correctly interpret a JSON message from the server telling them it is their turn. Otherwise, in later turns the player client should infer it is their turn from the turn order of the game. Controller and interaction model should be set to active state. If there is at least one info token in the pool, the give colour info action button should be interactable. All other player action buttons (from section 3.2.1 , use cases 4, 6, and 7) should also be clickable so the player may switch their state to those actions. After selecting the give colour info action, if the active player hovers their cursor over another player's cards, that card and all others in that player's hand matching that colour should be raised. If the cursor leaves that card the cards should again be lowered.

Test 1: If the active player selects one of those cards from another player's hand to inform about, that player will receive a message from the server indicating which cards match that colour. Those cards should be redrawn with the correct colour. Other players are also notified of the inform action and will flag the same cards with a colour token to indicate that the colour has been informed. The information token pool will be reduced by one in all client programs.

Test 2: This test will be done through visual inspection of the UI elements throughout the use case, ensuring interactability and visibility changes at each point described. Because it is a MVC system, seeing the cards change in the view should confirm that the model was successfully updated as expected.

Use Case 7: Give Number Info (Requirements Document p. 23)

If this is the first turn of the game the player should receive and correctly interpret a JSON message from the server telling them it is their turn. Otherwise, in later turns the player client should infer it is their turn from the turn order of the game. Controller and interaction model should be set to active state. If there is at least one info token in the pool, the give number info action button should be interactable. All other player action buttons (from section 3.2.1, use cases 4, 6, and 7) should also be clickable so the player may switch their state to those actions. After selecting the give number info action, if the active player hovers their cursor over another player's cards, that card and all others in that player's hand matching that number should be raised. If the cursor leaves that card the cards should again be lowered.

Test 1: If the active player selects one of those cards from another player's hand to inform about, that player will receive a message from the server indicating which cards match that number. Those cards should be redrawn with the correct number. Other players are also notified of the inform action and will flag the same cards with a number token to indicate that the number has been informed. The information token pool will be reduced by one in all client programs.

Test 2: This test will be done through visual inspection of the UI elements throughout the use case, ensuring interactability and visibility changes at each point described. Because it is a MVC system, seeing the cards change in the view should confirm that the model was successfully updated as expected.

Use Case 8: Look at Discard Pile (Requirements Document p. 23)

Testing this will require to view the UI as additions are made to the discard pile. The screen will close on user input.

Test 1: The user discards a card. This will update the discard pile with the new card added. The newly discarded card should be visible in the Discard Pile Object

Test 2: A player, separate from the user, discarding a card will update the discard pile with the new card added. The newly discarded card should be visible in the Discard Pile Object

Test 3: The View Discard Pile button should be clickable from all 6 game states.

- Active
- Active and Discard a Card selected
- Active and Play a Card selected
- Active and Give colour info

- Active and Give number info
- Inactive

Use case 9: Player Disconnects (Requirements Document p. 24)

Tests how the system responds to a player disconnecting from the game.

Test 1: A player separate from the user disconnects from the game. This should show the game end screen to the user

Use Case 10: Player exits the game (Requirements Document p. 25)

Tests whether the program exits when the exit button is pressed.

Test 1: The user presses the exit (X) button. The program should exit.

Integration Tests

Integration tests for Use Case 1: Hosting a game

Applicable to the Software Requirement Specification section: 3.2.1.1

1-2. Verify that clicking 'Host Game' button brings up field boxes for timeout limit, NSID, players, token. Player should still be able to click 'Join Game' instead.
- visual confirmation

3-5. After the user enters appropriate info into fields and clicks ok, verify that the connection is established with the server and JSON is marshalled with correct information and sent to create the game.
- assert that the JSON message contains the information entered by the user and that the connection is valid

6. Verify that the game was correctly created on the server.
- visual confirmation when client window shows a popup

7. Verify that the user has joined game and is presented the game signature to be used to join it, and how many more players needed.
- visual confirmation from same popup as above step

8. Verify that the GameModel, GameView, GameController, and GameInteraction model have been instantiated with the correct information.

- visual confirmation that elements look as they should, initial hands dealt by the server are correct on all clients

Integration tests for Use Case 2: Joining a Game

Applicable to the Software Requirement Specification section: 3.2.1.2

1-2. Verify that clicking the 'Join Game' button brings up field boxes for game-id and token. The player should still be able to click 'Create Game' instead.

- visual confirmation

3-4. After the user enters appropriate info into fields and clicks ok, verify that the connection is established with the server and JSON is marshalled with correct information and sent to join the game.

- assert that the JSON message contains the information entered by the user and that the connection is valid

5. Verify that the user has joined game and is presented the game signature to be used to join it, and how many more players needed.

- visual confirmation from same popup as above step

6. Verify that the GameModel, GameView, GameController, and GameInteraction model have been instantiated with the correct information.

- visual confirmation that elements look as they should, initial hands dealt by the server are correct on all clients

Integration tests for Use Case 3: Reading the rules

Applicable to the Software Requirement Specification section: 3.2.1.3

1-2. Verify that when the Rules button is clicked, the rules window is opened.

- visual confirmation.

3-4. Verify that when the different Rules section tabs are clicked that the appropriate information is shown.

- visual confirmation.

5-6. Verify that the discard pile window can be closed and re-opened.

- visual confirmation

Integration tests for Use Case 4-7: Play a Card, Discard a Card, Give Colour Info, Give Number Info

Applicable to the Software Requirement Specification sections: 3.2.1.4 to 3.2.1.7

1. Verify that all clients correctly determine whose turn it is from turn order and JSON interpretation.
    - visual confirmation as it requires syncing of multiple clients that don't know about each other.

2. Verify that controller and interaction model are both set to active state for the current active player and that other players are inactive.
    - visual confirmation
    - can also assert and test that way

3. Verify that the view draws the correct buttons to be interactable at the correct times based on model and interaction model updates, ie discard card is available only when a discard action can be taken, inform buttons only active when inform actions can be taken.
    - visual confirmation, and asserts on state changes

4. Verify that each action produces the proper changes in the model, ie. discard pile is added to when card is discarded or play fails.
    - assert that the correct method is called when it should be, methods themselves should be tested in Unit Testing

5. Verify that all clients maintain the same hands (exempting things unknown about their own hands, although given info should be maintained as well) and game information like tokens available
    - visual confirmation - if the view draws it that should be proof the model contains that information
    - can also use asserts that fields contain the right values after actions taken

6. Verify that the game ends properly as a result of: final fuse going off from failed play, final firework being built, final player taking their last turn when no cards can be drawn.

Integration tests for Use Case 8: Look at Discard Pile

Applicable to the Software Requirement Specification sections: 3.2.1.8

Test 1: Verify that when the discard pile button is clicked, the View brings up the discard pile window, with the information of the discard pile's contents correctly displayed.

- visual confirmation.

Test 2: When a card is discarded by a player, either by discarding or via a failed attempt to play the card, the discard pile object should be updated with the new information. Check to see that the discard pile contains the card that was discarded, in the appropriate amount. Check that the discard pile view window updates when the action is complete.

- assert that discard pile size is increased using a mock. Check that the contents match expected contents.

Test 3: When the user discards or fails to play a card, ensure that the information returned by the server is passed on to the discard pile, and ensure that the card is added to the discard pile. Check to see that the discard pile window updates when the action is complete.

- assert that discard pile size is increased using a mock. Check that the contents match expected contents.

Test 4: After any game view state change, the discard pile button should still be usable.

- assert that the discard pile button remains clickable.

Integration tests for Use Case 9: Player Disconnects

Applicable to the Software Requirement Specification section: 3.2.1.9

1. When a the server informs the player that another player has disconnected ending the game, display the Game End screen with score.

- visual check to ensure that the game end screen is displayed correctly.

2. Ensure that the game has been successfully closed and cleaned up.

- validate that the game model, view, and controller have been cleaned up.

3-4. Verify that the end game screen is shown to all users, ensure that the user can return to the main menu from the game end screen.

- visual check, ensure that the menu is visible and usable.

Integration tests for Use Case 10: Player Exits program

Test 1: When the user closes the game window, the game should close down.

- visual check. Ensure that all game windows close down when the main game window is closed.

Unit Tests

Most of these unit tests will be handled through main() method tests and JUnit testing ensuring that outputs of various functions match expected values given an assortment of valid inputs, effectively using very small scale mocks. The tests should cover all situations that can be reasonably expected to arise. For testing views, many tests will be manual visual confirmation, again using small scale mocks to draw an object and ensure it looks as it should. For the AI, much of the testing will need to be done through game situation mocks like stacked decks to test expected behaviour.

GameMenu class unit testing from Design Document

contructor GameMenu()

- visually verify that the menu was constructed.

method createGame()

- verify that that connections can be made with the server.
- verify that a game can be created.
- verify that aModel, iModel, aView, aController, game-id and token were all instantiated.
- verify that the text boxes are properly read.

method joinGame()

- verify that that connections can be made with the server.
- verify that aModel, iModel, aView, aController were all instantiated.

method howToPlay()

- verify that the rules windows is presented when pressed.

method exit()

- verify that the program is terminated.

GameController class unit testing from Design Document

constructor GameController()

- verify that a model, a view and an interaction model were all populated with the correct information

method handleMessage(String jsonMessage)

- verify that messages from the server are received, and that parseJSON is called, as well as the necessary functions based on the message.

method parseJSON(String jsonMessage)

- verify that the JSON message is parsed properly.

method sendJSON(Collection move)

- verify that the JSON messages are properly constructed and sent to the server.

GameModel class unit testing from Design Document

constructor GameModel(int timeout)

- verify that timeLimit variable is set according to timeout. Incorrect values should be impossible since timeout should be given by the server.

method dealTable(int playerCount, Collection startingHands)

- verify that cards dealt are correct for the number of players (4 for four or five players, 5 for two or three players).
- verify that the cards in each hand are in the right locations.
- check that tokens and fuses have been initialized to the correct values (3 and 8 respectively)

method addToken()

- verify that the token count is increased by one.
- verify that the token count cannot move above 8.

method removeToken()

- verify that the token count is decremented by one.
- verify that the token count cannot move below 0.

method getInfoTokens()

- verify that the method returns the correct number of information tokens available.

method removeFuse()

- verify that fuse count is decremented by one.
- verify that if fuse count reaches zero, game end is triggered.

method getFuses()

- verify that the number returned is the current number of fuses remaining.

method giveCard(String newCard)

- verify that the new card given by the string newCard is created and placed in the first available position in the current player's hand.
- verify that the number of remaining cards in the deck is decremented by one.
- verify that a card without information is created in the user's hand when the game deals a card to the user.

method discardCard(int player, int handPosition)

- verify that the card is removed from the correct location in the correct hand
- verify that the card removed is placed in the Discard Pile object
- verify that addToken() is called

method informCard(int player, char info)

- verify that the correct player is targeted
- verify that each card with the given information in that player's hand is marked as known
- verify that removeToken() is called

method playCard(int player, int handPosition)

- verify that the correct player is chosen and that the correct card is removed from their hand
- verify that the game adds a card to the appropriate fireworks stack if the card can be added
- verify that the game burns a fuse and sends the chosen card to the discard pile if the card cannot be added to the fireworks stack
- verify that the replacement card given by the server is placed into the player's hand via giveCard()

method getFireworkHeight()

- verify that the correct height is returned for the given firework colour

method nextTurn()

- verify that the current player counter is incremented.
- verify that when the new turn counter is moved to above the current number of players, the turn counter is reset to 1.

method currentTurn()

- verify that the number returned is the number of the player whose turn it is.

method getTime()

- verify that the remaining time for the current turn is accurately returned based on the current game's timeout.

method playerSeat()

- verify that the method returns the correct player seat number for the user's current game.

GameView class testing from Design Document

constructor GameView()

- verify that the view window is created and all known info is drawn correctly

method setModel(GameModel aModel)

- verify that the model is successfully set to the given GameModel

method setInteractionModel(GameInteractionModel iModel)

- verify that the interaction model is successfully set to the given GameInteractionModel

method drawTable()

- verify that the drawTable method calls the correct methods and all cards, hands, and UI elements are drawn correctly based on the current objects in the model

method drawHand(Hand hand)

- verify that this method takes a hand and draws it as expected

method drawCard(Card card)

- verify that this method takes a card and draws it as expected

method drawInfoTokens()

- verify that this method draws the correct number of info tokens as represented in the model, in the correct position and shape

method drawFireworks()

- verify that this method draws the correct firework stacks from information in the model, in the expected position

method drawFuses()

- verify that this method draws the correct amount of fuses from the model, in the position expected

method drawStatus()
- verify that this method draws the status bar with the correct time and number of cards left in the deck

Card class unit testing from Design Document

constructor Card(char number, char colour)
- verify that a valid object is instantiated whether it is given no arguments, number and null, null and colour, or both. Test the values of the initialized fields in each case.

method getNumber()
- verify that calling the method returns the card's rank or number

method getColour()
- verify that calling the method returns the card's suit or colour

method setNumber(char number)
- verify that calling the method correctly sets the card's number field to the number provided

method setColour(char colour)
- verify that calling the method correctly sets the card's colour field to the colour provided

method checkNumberKnown()
- verify that calling the method returns the boolean in the numberKnown field

method checkColourKnown()
- verify that calling the method returns the boolean in the colourKnown field

method numberGiven()
- verify that calling the method results in the numberKnown field being set to True for the card object

method colourGiven()
- verify that calling the method results in the colourKnown field being set to True for the card object

Hand class unit testing from Design Document

constructor Hand(int handSize)

- verify that the collection is of the correct size

method addCard(char colour, char number)

- verify that the new card is added to the collection in the first available position
- verify that the new card is accurate
- verify that a new card without information will be created if no arguments are given

method removeCard(int position)

- verify that the correct card is removed from the collection
- verify that there is an empty space left in the correct position in the hand

method informColour(char colour)

- verify that each card in the hand with a matching colour is marked as known

method informNumber(char number)

- verify that each card in the hand with a matching number is marked as known

Fireworks class unit testing from Design Document

constructor Fireworks()

- verify that the collection is instantiated with the height value for each firework set to 0

method addFirework(char colour)

- verify that any firework is properly incremented by 1 when its colour is given as an argument

method checkBuildValid(Card tryMe)

- verify that plays are accurately determined as valid when they should be, that the card is actually playable

method getFireworks()

- verify that the collection in the fireworks field is successfully returned and can be accessed

Table class unit testing from Design Document

constructor Table(int playerCount)

- verify that the correct number of hands are created based on the given number of players

method giveCard(int playerID, char colour, char number)

- verify that a card was added to the correct player's hand.
- verify that a card without any information can be added if no colour and number arguments are given

method removeCard(int playerID, int cardPosition)

- verify that a card was removed from the correct player's hand and from the correct position
- verify that the position where the card was removed is now empty

method informCard(intplayerID, String type, char info)

- verify that the correct Hand method is called depending on the given "type" argument
- verify that the information is applied to the correct player's hand
- verify that the cards to be informed are marked with the correct known status

AIController class unit testing from Design Document

method checkAvailableMoves()

- use a mock to verify that canInform and canDiscard are set to the correct boolean for token values 0,1,7,8. It is not possible for both to be false.

method bestPlay()

- use a mock to verify the best play move is returned based on the conventions chosen and statistical weights given to each possible play.

method bestInform()

- use a mock to verify the best inform move is returned based on the conventions chosen and statistical weights given to each possible inform.

method bestDiscard()

- use a mock to verify the best discard move is returned based on the conventions chosen and statistical weights given to each possible discard.

method performBestMove()

- verify proper comparison each of the best moves and invoke the proper function to perform the move.

DiscardPile class unit testing from Design Document

constructor DiscardPile()

- verify that after being instantiated the discardCount field is set to 0 and the collection is created

method getDiscards()

- verify that the collection of discarded cards is returned and accessible

method addDiscard(char number, char colour)

- verify that the given card is successfully added to the discard pile

method getDiscardCount()

- verify that value of the discardCount field is returned

DiscardPileView class unit testing from Design Document

constructor DiscardPileView()

- verify that the view is correctly instantiated and a separate window opens with the discard pile visible

method setModel(GameModel model)

- verify that the model field for the object is successfully set to the GameModel

method drawCard()

- verify that a card can be drawn with the expected appearance

method drawDiscardPile()

- verify that all cards are drawn in their expected position in the grid with the correct number of each

EndGameView class unit testing from Design Document

constructor EndGameView(Collection endInfo)

- verify that the view is correctly instantiated and game window changes to new view.
- visual confirmation.

method display()

- verify congratulatory message, score and graphical display of fireworks is drawn.
- visual confirmation.

method closeGame()

- verify endGame screen properly closes and user is returned to main menu.