

SFWRENG 3XA3: Test Plan

Lab 2 Group 7, Rummy For Dummies

Joy Xiao, xiaoz18

Benson Hall, hallb8

Smita Singh, sings59

April 09, 2021

Contents

1	General Information	1
1.1	Purpose	1
1.2	Scope	1
1.3	Acronyms, Abbreviations, and Symbols	1
1.4	Overview of Document	2
2	Plan	2
2.1	Software Description	2
2.2	Test Team	2
2.3	Automated Testing Approach	3
2.4	Testing Tools	3
2.5	Testing Schedule	3
3	System Test Description	3
3.1	Tests for Functional Requirements	3
3.1.1	Card Tests	3
3.1.2	Discard Pile Tests	5
3.1.3	Stock Pile Tests	6
3.1.4	Computer Tests	6
3.1.5	Hand Tests	9
3.1.6	Player Tests	11
3.1.7	Meld Tests	15
3.1.8	UserInputOps Tests	17
3.1.9	GameOps Tests	19
3.2	Tests for Nonfunctional Requirements	24
3.2.1	Look and Feel Requirements	24
3.2.2	Usability and Humanity Requirements	24
3.2.3	Performance Requirements	24
3.2.4	Operational and Environmental Requirements	25
3.2.5	Maintainability and Support Requirements	25
3.2.6	Cultural Requirements	25
3.2.7	Legal Requirements	26
3.3	Traceability Between Test Cases and Requirements	27
4	System Tests for Proof of Concept	29

5	Comparison to Existing Implementation	32
6	Unit Testing Plan	32
6.1	Unit testing of internal functions	32
6.2	Unit testing of output files	32
7	Appendix	33

List of Tables

1	Revision History	ii
2	Table of Abbreviations	1
3	Table of Definitions	2
4	Traceability Matrix: Functional Requirement	27
5	Traceability Matrix: Non-Functional Requirement	29

List of Figures

Table 1: **Revision History**

Date	Version	Notes
February 26, 2021	1.0	Started on test plan
March 5, 2021	1.1	Finished the test plan and matrix
April 09, 2021	2.0	Finished test plan revision 1

1 General Information

1.1 Purpose

The purpose of the document is to outline the testing plan for Rummy for Dummies. The test cases will cover all functional and non-functional requirements, and elaborate on the details behind the tests being written and executed. Through the test cases, confidence can be achieved that the game can be played without error.

1.2 Scope

The test plan provides the space for testing the functional and non-functional requirements of Rummy for Dummies. The objective of the test plan is to prove that a full game of Gin-Rummy can be played without any error. The game will be able to be played with valid inputs from the user and any invalid inputs will be handled appropriately.

Additionally, the test plan will act as a schedule for all test-based activities. The test plan will outline all testing methods and tools that will be used.

1.3 Acronyms, Abbreviations, and Symbols

Table 2: Table of Abbreviations

Abbreviation	Definition
PoC	Proof of Concept
CLI	Command Line Interface

Table 3: **Table of Definitions**

Term	Definition
Automated Testing	Testing that uses software separate to the software being tested to handle execution of tests
Manual Testing	Testing that requires human intervention or input to handle execution of tests
Statement Coverage	A test suite achieves statement coverage if it results in every statement in the program to be run at least once
Branch Coverage	A test suite achieves branch coverage if it results in every branch being taken at least once
Condition Coverage	A test suite achieves condition coverage if it results in each condition in each branching instruction being both true and false

1.4 Overview of Document

In the document, Section 2 will detail the plan for future tests on Rummy for Dummies. Section 3 will describe all tests that will be conducted on the system. Section 4 will discuss tests for the PoC. Section 5 will discuss the tests that will be used to compare between Rummy for Dummies and the original source code project. Section 6 will explain the unit testing plan.

2 Plan

2.1 Software Description

The software is a single-player card game implementation of Gin-Rummy with a computer opponent. This game is implemented in Java.

2.2 Test Team

The individuals listed below will be responsible for writing and executing all test cases, and ensuring that Rummy for Dummies passes all these tests.

- Joy Xiao
- Benson Hall

- Smita Singh

2.3 Automated Testing Approach

Most tests performed on methods will be automated, while other tests such as integration tests will be manually conducted. In these integration tests, the JUnit assertion statements will not be utilized. Instead, testers will manually check printed results on the console/CLI for the correct output.

2.4 Testing Tools

All tests will be written using the JUnit framework.

2.5 Testing Schedule

The Gantt chart can be found [here](#)

3 System Test Description

3.1 Tests for Functional Requirements

3.1.1 Card Tests

1. FR-C-1: Test accessor for card suit

Type: Unit test, functional, dynamic, automated

Initial State: Card ~~card1~~ = new Card(Suit.D, 1);

Input: ~~card1~~ Card

Output: Suit.D

How test will be performed: Call Card.getSuit and see if it matches expected output

2. FR-C-2: Test ~~constructor with invalid rank. accessor for card rank~~

Type: Unit test, functional, dynamic, automated

Initial State: ~~Card card1 = new Card(Suit.D, -1)~~ N/a

Input: ~~card1~~ Card = new Card(Suit.D, -1)

Output: ~~Invalid Rank Exception~~ IllegalArgumentException

How test will be performed: ~~-Call Card.getRank and see if it matches expected output~~ The card constructor is called when the card is initialized.

3. FR-C-3: Test accessor for point value of card (I)

Type: Unit test, functional, dynamic, automated

Initial State: Card ~~card1~~ = new Card(Suit.D, 11);

Input: ~~card1~~ Card

Output: 10

How test will be performed: Call Card.points and see if it matches expected output

4. FR-C-4: Test accessor for point value of card (II)

Type: Unit test, functional, dynamic, automated

Initial State: Card ~~card1~~ = new Card(Suit.C, 2);

Input: ~~card1~~ Card

Output: 2

How test will be performed: Call Card.points and see if it matches expected output

5. FR-C-5: Test String representation of card (I)

Type: Unit test, functional, dynamic, automated

Initial State: Card ~~=card1~~ = new Card(Suit.H, 11);

Input: ~~card1~~ Card

Output: "Jh"

How test will be performed: Call Card.toString and see if it matches expected output

6. FR-C-6: Test String representation of card (II)

Type: Unit test, functional, dynamic, automated

Initial State: Card ~~=card1~~ = new Card(Suit.C, 2);

Input: ~~card1~~ Card

Output: "2c"

How test will be performed: Call Card.toString and see if it matches expected output

7. FR-C-7: Test Symbol representation of card

Type: Unit test, functional, dynamic, automated

Initial State: Card = new Card(Suit.S, 11)

Input: Card

Output: "J spade symbol"

How test will be performed: Call Card.toSymbol and see if it matches expected output

8. FR-C-8: Test getRank method

Type: Unit test, functional, dynamic, automated

Initial State: Card = new Card(Suit.D, 1)

Input: Card

Output: "1"

How test will be performed: Call Card.getRank and see if it matches expected output

3.1.2 Discard Pile Tests

1. FR-DP-1: Test display of discard pile

Type: Unit test, functional test, dynamic, automated

Initial State: DiscardPile = [5h, 9d]

Input: DiscardPile.displayTopCard

Output: 9h displayed

How test will be performed: Call DiscardPile.displayTopCard

2. FR-DP-2: Test drawing card from discard pile

Type: Unit test, functional test, dynamic, automated Initial State:
DiscardPile = [5h, 9d]

Input: DiscardPile.pop

Output: DiscardPile = [5h]

How test will be performed: Call DiscardPile.pop

3.1.3 Stock Pile Tests

1. FR-SP-1: Test drawing from stock pile with 1 or more cards

Type: Unit test, functional test, dynamic, automated

Initial State: StockPile = [As, 5d]

Input: Stockpile.pop

Output: Stock pile = [As]

How test will be performed: Call StockPile.pop

2. ~~FR-SP-2: Test drawing from stock pile with 0 cards~~

~~Type: Integration test, functional test, dynamic, automated~~

~~Initial state: StockPile = []~~

~~Input: Stockpile.pop~~

~~Output: Does not allow player to draw from the stock pile. No error thrown. User prompted to choose another selection.~~

~~How test will be performed: Call StockPile.pop~~

3.1.4 Computer Tests

1. FR-CP-1: Test computer make move method (I)

Type: Integration test, functional test, dynamic, ~~manual~~, automated

Initial state: DiscardPile = [4h, 9c, 5d], Computer Hand = [4d, 6d, Ks, Js, Ad, 2d], StockPile = [Qs, 3d]

Input: Computer.makeMove

Output: DiscardPile = [4h, 9c, ~~5d~~, Ks], Computer Hand = [4d, 6d, Js, Ad, 2d, 5d], StockPile = [Qs, 3d]. Returns False.

How test will be performed: Call Computer.makeMove method. ~~Manually check output~~

2. FR-CP-2: Test computer make move method (II)

Type: Integration test, functional test, dynamic, ~~manual~~, automated

Initial state: DiscardPile = [4h, Jc, 10d, Ks], Computer Hand = [4d, 6d, Js, Ad, 2d, 5d], StockPile = [Qs, 3d]

Input: Computer.makeMove

Output: DiscardPile = [4h, Jc, 10d, Ks, Js], Computer Hand = [4d, 6d, Ad, 2d, 5d, 3d], StockPile = [Qs]. Returns False.

How test will be performed: Call Computer.makeMove method. ~~Manually check output~~

3. FR-CP-3: Test computer make move method (III)

Type: Integration test, functional test, dynamic, ~~manual~~, automated

Initial state: DiscardPile = [4h, Jc, 10d, As], Computer Hand = [4d, 6d, Js, Ad, 2d, 5d], StockPile = [Qs, 3d]

Input: Computer.makeMove

Output: DiscardPile = [4h, Jc, 10d, Js], Computer Hand = [4d, 6d, Ad, 2d, 5d, As], StockPile = [Qs, 3d]. Returns False.

How test will be performed: Call Computer.makeMove method. ~~Manually check output~~

4. FR-CP-4: Test that computer never discards a meld card (I)

Type: Integration test, functional test, dynamic, ~~manual~~, automated

Initial state: DiscardPile = [4h, Jc, 10d, As], Computer Hand = [4d, 6d, Ad, 2d, Kc, Kd, Ks], StockPile = [Qs, 3d]

Input: Computer.makeMove

Output: DiscardPile = [4h, Jc, 10d, 6d], Computer Hand = [4d, Ad, 2d, Kc, Kd, Ks, As], StockPile = [Qs, 3d]. Returns False.

How test will be performed: Call Computer.makeMove. ~~Manual check printing out the computer's hand every turn and to make sure meld cards are never discarded~~

5. FR-CP-5: Test that computer never discards a meld card (II)

Type: Integration test, functional test, dynamic, ~~manual~~, automated

Initial state: DiscardPile = [4h, Jd, 10d, As], Computer Hand = [4d, 6d, Ad, 2d, Jc, Qc, Kc], StockPile = [Qs, 3d]

Input: Computer.makeMove

Output: DiscardPile = [4h, Jd, 10d, As, 6d], Computer Hand = [4d, Ad, 2d, Jc, Qc, Kc, As], StockPile = [Qs, 3d]. Returns False.

How test will be performed: Call Computer.makeMove. ~~Manual check printing out the computer's hand every turn and to make sure meld cards are never discarded~~

6. FR-CP-6: Test that computer knocks whenever it's hand has deadwood score of 10 or less

Type: Unit test, functional test, dynamic, automated

Initial state: DiscardPile = [], StockPile = [], Computer Hand = [4d, Ad, 2d, Jc, Qc, Kc, As]

Input: computer.makeMove

Output: Returns True

How test will be performed: Call computer.makeMove

7. FR-CP-7: Test computer move when discard pile has 1 card

Type: Unit test, functional test, dynamic, automated

Initial state: DiscardPile = [4h], StockPile = [Qs, 3d], Computer Hand = [4d, 6d, Js, Ad, 2d, 5d]

Input: computer.makeMove

Output: DiscardPile = [Js], StockPile = [Qs, 3d], Computer Hand = [4d, 6d, Ad, 2d, 5d, 4h]. Returns False.

How test will be performed: Call computer.makeMove

8. FR-CP-8: Test computer does not draw from empty stock pile

Type: Unit test, functional test, dynamic, automated

Initial state: DiscardPile = [4h], StockPile = [], Computer Hand = [4d, 6d, Js, Ad, 2d, 5d]

Input: computer.makeMove

Output: DiscardPile = [Js], StockPile = [], Computer Hand = [4d, 6d, Ad, 2d, 5d, 4h]. Returns False.

How test will be performed: Call computer.makeMove

3.1.5 Hand Tests

1. FR-H-1: Test remove card from hand (I)
Type: Unit test, functional test, dynamic, automated
Initial state: hand = [As, 2s, Qs, 3d, 5d]
Input: "As"
Output: hand = [2s, Qs, 3d, 5d]
How test will be performed: Call Hand.remove
2. FR-H-2: Test remove card from hand (II)
Type: Unit test, functional test, dynamic, automated
Initial state: hand = [As, 2s, Qs, 3d, 5d]
Input: "Kc"
Output: hand = [As, 2s, Qs, 3d, 5d]
How test will be performed: Call Hand.remove
3. FR-H-3: Test remove card from hand (III)
Type: Unit test, functional test, dynamic, automated
Initial state: hand = [As, 2s, Qs, 3d, 5d]
Input: "AS"
Output: hand = [2s, Qs, 3d, 5d]
How test will be performed: Call Hand.remove
4. FR-H-4: Test remove card from hand (IV)
Type: Unit test, functional test, dynamic, automated
Initial state: hand = [As, 2s, Qs, 3d, 5d]
Input: "KS"
Output: hand = [As, 2s, Qs, 3d, 5d]
How test will be performed: Call Hand.remove

5. FR-H-5: Test displaying of hand
Type: Unit test, functional test, dynamic, manual
Initial state: Player object initialized with a non-empty hand
Input: Hand
Output: All cards in player's hand is displayed correctly
How test will be performed: Call Hand.display
6. FR-H-6: Test contains method (I)
Type: Unit test, functional test, dynamic, automated
Initial state: hand = [As, 2s, Qs, 3d, 5d]
Input: input = "As"
Output: Return True
How test will be performed: Call Hand.contains
7. FR-H-7: Test contains method (II)
Type: Unit test, functional test, dynamic, automated
Initial state: hand = [As, 2s, Qs, 3d, 5d]
Input: input = "Ks"
Output: Return False
How test will be performed: Call Hand.contains
8. FR-H-8: Test contains method (III)
Type: Unit test, functional test, dynamic, automated
Initial state: hand = [As, 2s, Qs, 3d, 5d]
Input: input = "AS"
Output: Return True
How test will be performed: Call Hand.contains
9. FR-H-9: Test contains method (IV)
Type: Unit test, functional test, dynamic, automated
Initial state: hand = [As, 2s, Qs, 3d, 5d]

Input: input = "KS"

Output: Return False

How test will be performed: Call Hand.contains

3.1.6 Player Tests

1. FR-P-1: Test accessor for player's name

Type: Unit test, dynamic, automated

Initial state: Player p = new Player("P1");

Input: p

Output: "P1"

How test will be performed: Call Player.getName and check that the name matches

2. FR-P-62: Test adding a card to player's hand

Type: Unit test, dynamic, automated

Initial state: p.hand = [~~Ah, Ae, 4d, 5s, 10h~~][Ah, 2s, Qs, 3d, 5d];

Input: p, new Card(Suit.S, 132)

Output: p.hand = [~~Ah, Ae, 4d, 5s, 10h, Qs~~][Ah, 2s, Qs, 3d, 5d, Ks];

How test will be performed: Call Player.addCardToHand method with the Card object as a parameter. Call Player.getHand to verify that the new card has been added to the hand

3. FR-P-23: Test accessor for player's hand

Type: Unit test, dynamic, automated

Initial state: p.hand = [Ah, 2s, Qs, 3d, 5d];

Input: p

Output: hand = [Ah, 2s, Qs, 3d, 5d]

How test will be performed: Call Player.getHand and check that assigned hand matches received object

4. FR-P-~~34~~: Test accessor for player's total score
 Type: Unit test, dynamic, automated
 Initial state: `p.totalScore = 12;`
 Input: `p`
 Output: `totalScore = 12`
 How test will be performed: Call `Player.getTotalScore` and check that assigned score matches score received from call

5. FR-P-~~45~~: Test accessor for player's deadwood score
 Type: Unit test, dynamic, automated
 Initial state: `p.hand = [Ah, Ac, Ad, As, 2s, Jh, Qs, 9h][Ah, Ac, Ad, As, 2s, 2c, 2d, 3d, Qs, 9h]`
 Input: `p`
 Output: `deadwoodScore = 2 + 10 + 10 + 9 = 31 3 + 10 + 9 = 22`
 How test will be performed: Call `Player.recalculateDeadwoodScore` to calculate the deadwood score. Call `Player.getDeadwoodScore` method and check that the pre-calculated deadwood score matches the deadwood score received from call

6. FR-P-~~56~~: Test accessor for player's melds
 Type: Unit test, dynamic, automated
 Initial state: `p.hand = [Ah, Ac, Ad, As, 2s, 2d, 2c, 3ds, Qs, 9h];`
 Input: `p`
 Output: `melds = [[As, Ah, Ac, Ad],[2s, 2d, 2c]]`
 How test will be performed: Call `Player.checkMelds` to calculate melds in the hand, then call `Player.getMelds` method and check that the melds yielded from the calculation are correct

7. FR-P-7: Test discarding a card that exists in the player's hand
 Type: Unit test, dynamic, automated
 Initial state: `p.hand = [4d, 8s, Jh][Ah, 2s, Qs, 3d, 5d]`
 Input: `p.hand, 5d "jh"`

Output: p.hand = ~~[4d, 8s]~~[Ah, 2s, Qs, 3d], new Card(Suit.DH, 5H) is returned.

How test will be performed: Call Player.discardFromHand with a string representation of the card to discard. Verify that the card has been returned and that the hand no longer contains this card

8. FR-P-8: Test discarding a card that does not exist in the player's hand

Type: Unit test, dynamic, automated

Initial state: p.hand = ~~[4d, 8s, Kh, Qs, As]~~[Ah, 2s, Qs, 3d, 5d]

Input: p, "jh"

Output: ~~p.hand = [4d, 8s, Kh, Qs, As], null returned~~
~~IllegalArgumentException~~

How test will be performed: Call Player.discardFromHand with a string representation of the non-existent card to discard. Verify that ~~a null output is returned, and the hand is unchanged-an~~ exception is raised from this call.

9. FR-P-9: Test adding a score to the total score

Type: Unit test, dynamic, automated

Initial state: p.totalScore = 120;

Input: p, 15

Output: p.totalScore = 275

How test will be performed: Call Player.getTotalScore to get the original score of 120. Call Player.addToTotalScore with 15 as the parameter. Call Player.getTotalScore to get the new score. Verify that the original score added to the integer results in the new score, that is, 120 + 15 = 275.

10. FR-P-10: Test getting deadwood cards from the player's hand

Type: Unit test, dynamic, automated

Initial state: p.hand = [Ah, Ac, Ad, 2s, 9hs, 10h, Jd]

Input: p

Output: deadwoodCards = [2s, 9hs, 10h, Jd]

How test will be performed: Call Player.extractDeadwood

11. FR-P-11: Test recalculation of deadwood score

Type: Unit test, dynamic, automated

Initial state: $p.hand = [Ah, Ac, Ad, 2s, 9s, 10h, Jd]$, $p.deadwoodScore = 045$

Input: p

Output: $p.deadwoodScore = 2 + 9 + 10 + 10 = 31$

How test will be performed: Call `Player.recalculateDeadwoodScore` to calculate the deadwood score. Call `Player.getDeadwoodScore` method and check that the pre-calculated deadwood score matches deadwood score received from call

12. FR-P-12: Test accessor for player's melds after checking for melds

Type: Integration test, dynamic, automated

Initial state: $p.hand = [As, 2s, 3s, 4d, 5d, 6d, 7d, 10h]$, $p.melds = [[As, 2s, 3s]]$

Input: p

Output: $p.melds = [[As, 2s, 3s], [4d, 5d, 6d, 7d]]$

How test will be performed: Call `Player.checkMelds` to calculate melds in the hand, then call `Player.getMelds` method and check that the melds yielded from the calculation are correct

13. FR-P-13: Test resetting hand

Type: Unit test, dynamic, automated

Initial state: $p.hand = [Ah, Ac, Ad, 2s, 9h, 10h, Jd][Ah, 4d, 3s, 9d]$

Input: p

Output: $p.hand = []$

How test will be performed: Call `Player.resetHand` to reset the hand. Call `Player.getHand` to verify that the hand is an empty list

14. FR-P-14: Test resetting deadwood score

Type: Unit test, dynamic, automated

Initial state: $p.deadwoodScore = 3145$

Input: p

Output: `p.deadwoodScore = 0`

How test will be performed: Call `Player.resetDeadwoodScore` to reset the deadwood score. Call `Player.getDeadwoodScore` to verify that the deadwood score is zero

15. FR-P-15: Test resetting list of melds

Type: Unit test, dynamic, automated

Initial state: `p.melds = [[Ah 3s, Ad 4s, Ac 5s]]`

Input: `p`

Output: `p.melds = []`

How test will be performed: Call `Player.resetMelds` to reset list of melds. Call `Player.getMelds` to verify that the list of melds is empty

16. FR-P-16: Test resetting total score

Type: Unit test, dynamic, automated

Initial state: `p.totalScore = 12`

Input: `p`

Output: `p.totalScore = 0`

How test will be performed: Call `Player.resetTotalScore` to reset the total score. Call `Player.getTotalScore` to verify that the total score is now 0

3.1.7 Meld Tests

1. ~~FR-M-1: Test sequence melds with minimum meld cards~~

~~Type: Unit test, dynamic, functional test, automated~~

~~Initial state: Player hand has 11 cards~~

~~Input: Input Hand = Hand with Cards [3h, 7d, Qh, 4h, Ke, Kd, Ae, Ad, 5h, 6e, 6d]~~

~~Output: InvalidHand Exception~~

~~How test will be performed: Call Meld.checkMelds~~

2. FR-M-~~21~~: Test sequence melds with more than 3 meld cards
 Type: Unit test, dynamic, functional test, automated
 Initial state: Player hand has 4 cards in consecutive rank with same suit eg. 3S, 4S, 5S, 6S
 Input: Hand with Cards: [3s, 7d, Qh, 4s, Kc, Kd, Ac, Ad, 5s, 6s]
 Output: [[3S, 4S, 5S, 6S]]
 How test will be performed: Call Meld.checkMelds
3. FR-M-~~32~~: Test melds with 3 cards of same rank
 Type: Unit test, dynamic, functional test, automated
 Initial state: Player's hand has 3 cards of same rank
 Input: Hand with Cards: [3s, 7d, Qh, 4s, Kc, Kd, Ac, Ad, Ah, 6s]
 Output: [~~Ae, Ad, Ah,~~] [Ah, Ac, Ad]
 How test will be performed: Call Meld.checkMelds
4. FR-M-~~43~~: Test melds with 4 cards of same rank
 Type: Unit test, dynamic, functional test, automated
 Initial state: Player's hand has 4 cards of same rank
 Input: Hand with Cards: [Kd, 5d, 4d, 5c, Jc, Qd, 5h, 5s, 3s, Js]
 Output: [~~5h, 5s, 5d, 5c,~~] [5h, 5s, 5c, 5d]
 How test will be performed: Call Meld.checkMelds
5. FR-M-~~54~~: Test meld with 2 sequence melds and one group meld
 Type: Unit test, dynamic, functional test, automated
 Initial state: Player's hand has 2 sequence melds and one group meld
 Input: Hand with Cards: [3s, 4s, 5s, Kc, Jc, Qc, Ac, Ad, Ah, As]
 Output: [[3s, 4s, 5s], [Jc, Kc, Qc], ~~[Ac, Ad, Ah, As]~~ [Ah, As, Ac, Ad]]
 How test will be performed: Call Meld.checkMelds
6. FR-M-~~65~~: Test meld with 1 sequence and 1 group meld
 Type: Unit test, dynamic, functional test, automated
 Initial state: Player's hand has 1 sequence melds and 1 group meld

Input: Hand with Cards: [10c,Js, Ks, 10d, Qs, 10h, Ac, 4c, 6h, 9h]

Output:[[Js,Qs,Ks],[10e,10d,10h][10h,10c,10d]]

How test will be performed: Call Meld.checkMelds

7. FR-M-76: Test Melds with overlapping cards

Type: Unit test, dynamic, functional test, automated

Initial state: Player's hand has a meld that can belong to a sequence and group meld

Input: [Ac, Ad, Ah, As,2s,3s,Ks,Jh, Jd, 7s]

Output: [[As,2s,3s],[Ac,Ad,Ah][Ah,Ac,Ad]]

How test will be performed: Call Meld.checkMelds

3.1.8 UserInputOps Tests

1. ~~FR-UIO-1: Test that user can only discard cards that are in their hand~~

~~Type: Unit test, dynamic, manual~~

~~Initial state: (Player) p.hand = [Ah, 4d, 5d, 8s, 10e]~~

~~Input: p, User inputs: "9d", "2s", "10e"~~

~~Output: "10e"~~

~~How test will be performed: Call UserInputOps.chooseDiscard method and enter input to test discard choice when prompted by the scanner —only cards that exist in the player's hand will terminate the test successfully~~

2. FR-UIO-12: Test that user inputting valid inputs for deciding on knocking results in successful termination

Type: Unit test, dynamic, combinatorial coverage-based, manual

Initial state: N/A

Input: User input: 'Y'

Output: 'y'

How test will be performed: Call UserInputOps.knock method and enter the valid input when prompted by the scanner

3. FR-UIO-**23**: Test that user inputting invalid inputs for deciding on knocking is handled properly
 Type: Unit test, dynamic, partition-based, fuzz testing, manual
 Initial state: N/A
 Input: User inputs: 'h', 'e', 'l', 'l', 'o', "othello", "No"
 Output: 'n'
 How test will be performed: Call UserInputOps.knock method and enter a series of invalid inputs when prompted by the scanner. Termination should not occur until a valid input is received
4. FR-UIO-**34**: Test that user inputting valid inputs for deciding on playing a new game of Gin-Rummy results in successful termination
 Type: Unit test, dynamic, combinatorial coverage-based, manual
 Initial state: N/A
 Input: User input: 'Y'
 Output: 'y'
 How test will be performed: Call UserInputOps.playAgain method and enter a valid input when prompted by the scanner
5. FR-UIO-**45**: Test that user inputting invalid inputs for deciding on playing a new game of Gin-Rummy is handled properly
 Type: Unit test, dynamic, partition-based, fuzz testing, manual
 Initial state: N/A
 Input: User inputs: "To", "be", "or", "not", "to", "Be", "yEs"
 Output: 'n' - note that 'not' begins with n
 How test will be performed: Call UserInputOps.playAgain method and enter a series of invalid inputs when prompted by the scanner. Termination should not occur until a valid input is received
6. FR-UIO-**56**: Test/Simulate user making a valid decision in a single round
 Type: Unit test, dynamic, combinatorial coverage-based, manual
 Initial state: N/A

Input: 3

Output: 3

How test will be performed: Call `UserInputOps.playerDecision` method and enter a valid input

7. FR-UIO-67: Test that when the user making a invalid decision in a single round, it is handled appropriately

Type: Integration test, dynamic, partition-based, fuzz testing, manual

Initial state: N/A

Input: -1, 45, "hello", 'y', "tomorrow and tomorrow", 2

Output: 2

How test will be performed: Call `UserInputOps.playerDecision` method and enter a series of invalid inputs when prompted by the scanner. Test should not terminate successfully until a valid input is received.

8. FR-UIO-78: Test username is received properly

Type: Unit test, functional, dynamic, manual

Initial state: N/A

Input: "AC"

Output: "AC"

How test will be performed: Call `UserInputOps.username` method and enter username when prompted by the scanner

3.1.9 GameOps Tests

1. FR-GO-1: Test that score is calculated correctly and given to the right player (I)

Type: Unit test, dynamic, automated

Initial state: (Player) `p1.deadwoodScore = 10`, `p1.totalScore = 0`, (Player) `p2.deadwoodScore = 15`, `p2.totalScore = 0`

Input: `p1`, `p2`

Output: `p1.totalScore = 5`, `p2.totalScore = 0`

How the test will be performed: Call `GameOps.calculateScore` method with player deadwood score requirements

2. FR-GO-2: Test that score is calculated correctly and given to the right player (II)
Type: Unit test, dynamic, automated
Initial state: p1.deadwoodScore = 20, p1.totalScore = 0, p2.deadwoodScore = 9, p2.totalScore = 0
Input: p1, p2
Output: p1.totalScore = 0, p2.totalScore = 11
How the test will be performed: Call GameOps.calculateScore method with player deadwood score requirements
3. FR-GO-3: Test that going Gin works correctly
Type: Unit test, functional, dynamic, automated
Initial state: p1.deadwoodScore = 0, p1.totalScore = 0, p2.deadwoodScore = 15, p2.totalScore = 0
Input: p1, p2
Output: p1.totalScore = 35, p2.totalScore = 0
How the test will be performed: Call GameOps.calculateScore method with player deadwood score requirements
4. FR-GO-4: Test that stock pile is created correctly Type: Unit test, functional, dynamic, automated
Initial state: N/A
Input: N/A
Output: Stock Pile with 52 unique cards (A to K, all suits)
How the test will be performed: Call GameOps.createStockPile
5. FR-GO-5: Test that discard pile is created correctly Type: Unit test, functional, dynamic, automated
Initial state: N/A
Input: N/A
Output: Empty stack/discard pile
How the test will be performed: Call GameOps.createDiscardPile

6. ~~FR-GO-6: Test that card being discarded from player's hand ends up on top of the discard pile~~
~~Type: Unit test, functional, dynamic, automated~~
~~Initial state: (Player) p.hand = [Ah, Ac, 5d, Jc], (DiscardPile) dp.peek = Qs~~
~~Input: "Jc"~~
~~Output: dp.top = Jc, p.hand = [Ah, Ac, 5d]~~
~~How the test will be performed: Call GameOps.discardCard~~
7. ~~FR-GO-7: Test that user can only discard cards that are in their hand~~
~~Type: Integration test, functional, dynamic, manual~~
~~Initial state: p.hand = [Ah, Ac, 5d, Jc], dp.peek = Qs~~
~~Input: User inputs: "2s", "3d", "4s", "5d"~~
~~Output: p.hand = [Ah, Ac, Jc], dp.peek = 5d~~
~~How the test will be performed: Call UserInputOps.chooseDiscard to choose a card to discard. If the card does not exist in the player's hand, prompt again. After a valid input is received, GameOps.discardCard is called to discard the card.~~
8. FR-GO-68: Test that opening distribution of cards is done correctly
Type: Unit test, dynamic, automated
Initial state: StockPile sp = createStockPile();
Input: N/A
Output: Player and cpu hands of size 10, discard pile of size 1, stock pile contains the rest of the cards
How the test will be performed: Call GameOps.distributeCards with a brand new stock pile
9. FR-GO-79: Test that opening distribution of cards will only be done with a 52-card stock pile Type: Unit test, functional, dynamic, automated
Initial state: Stock pile of size 285
Input: N/A

Output: Rejected status

How the test will be performed: Call GameOps.distributeCards with the incorrect stock pile

10. ~~FR-GO-10: Test drawing from stock pile~~

~~Type: Unit test, dynamic, automated~~

~~Initial state: p.hand = [Ah, Ad, 3d, 5c], sp.empty == false~~

~~Input: N/A~~

~~Output: p.hand = [Ah, Ad, 3d, 5c, New Card()]~~

~~How the test will be performed: Though the stock pile cannot be viewed, its sizes can be compared to confirm change has occurred~~

11. ~~FR-GO-11: Test drawing from discard pile~~

~~Type: Unit test, functional, dynamic, automated~~

~~Initial state: p.hand = [Ah, Ad, 3d, 5c], dp.peek = Qd~~

~~Input: N/A~~

~~Output: p.hand = [Ah, Ad, 3d, 5c, Qd]~~

~~How the test will be performed: Call GameOps.drawFromDiscardPile method~~

12. FR-GO-~~812~~: Test interfacing playAgain method Type: Integration test, functional, dynamic, manual

Initial state: N/A

Input: User inputs: 'A', 'B', 'n'

Output: 'n'

How the test will be performed: Call GameOps.playAgain method and enter the set of inputs. Only the valid inputs will result in a successful termination

13. FR-GO-~~913~~: Test process decision method Type: Integration test, combinatorial coverage-based, dynamic, manual

Initial state: New game has started

Input: User's decision based on possible moves

Output: Depends on move made - 1 will draw a card from the stock pile, 2 will draw a card from the discard pile, 3 will show melds, 4 will show deadwood score and prompt user if they wish to knock. If user knocks, return true, else return false

How the test will be performed: Call GameOps.processDecision method.
The success of the test depends on what the user sees. No exceptions should occur, termination should be successful.

14. FR-GO-104: Test reset for a new deal Type: Integration test, functional, dynamic, automated

Initial state: p.hand = [Ah, Ac, Ad, 6s], p.deadwoodScore = 6, p.melds = [Ah, Ac, Ad]

Input: N/A

Output: p.hand = [], p.melds = [], p.deadwoodScore = 0

How the test will be performed: Call GameOps.resetEverything method

15. FR-GO-115: Test interfacing username method

Type: Integration test, dynamic, manual

Initial state: N/A

Input: "AC"

Output: "AC"

How test will be performed: Call GameOps.username method and enter username when prompted by the underlying Scanner

16. FR-GO-12: Test decision making when stock pile is empty

Type: Unit test, dynamic, manual

Initial state: (StockPile) sp = new StockPile()

Input: User inputs 1 when prompted

Output: User should not be allowed to draw from the stock pile and is prompted to make a new valid decision

How test will be performed: Call GameOps.processDecision and enter specified input

3.2 Tests for Nonfunctional Requirements

3.2.1 Look and Feel Requirements

NFR-LF-1: Test game appearance

Type: Unit test, non-functional, dynamic, manual

Initial state: Cards dealt, player's hand and discard pile displayed

Input: N/A

Output: Cards displayed have a simplistic look

How the test will be performed: Manually check the look of the cards

3.2.2 Usability and Humanity Requirements

NFR-UH-1: Test game Usability

Type: System test, non-functional, dynamic, manual

Initial state: Cards dealt, player's hand and discard pile displayed

Input: N/A

Output: Easily set up and play the game

How the test will be performed: Manually set up and play game to check the ease of use

3.2.3 Performance Requirements

NFR-P-1: Test performance speed

Type: System testing, non-functional, dynamic, manual

Initial state: Start a new game

Input: User interactions with system

Output: System should respond within 0.5 seconds of user input, according to the non-functional requirements

How the test will be performed: Play a round of the game manually and time the system response

NFR-P-2: Test game rules requirements

Type: System testing, non-functional, dynamic, manual

Initial state: N/A

Input: N/A

Output: Legal moves (according to game rules) can be done without errors. User's prompted for another input if they are making an illegal move

How the test will be performed: Manually play the game to make sure all legal moves can be made and any illegal moves are prevented

3.2.4 Operational and Environmental Requirements

NFR-OE-1: Test operational requirements

Type: System testing, non-functional, static, manual

Initial state: N/A

Input: N/A

Output: Game can be played on various operating systems

How the test will be performed: Manually test that the program can be operated on Windows, Linux, and MacOS operating system

3.2.5 Maintainability and Support Requirements

NFR-MS-1: Test maintainability requirements

Type: System testing, non-functional, manual

Initial state: N/A

Input: N/A

Output: Well documented source code as well as game rules and how to run the game for users

How the test will be performed: Manually test that the program is well documented

3.2.6 Cultural Requirements

NFR-C-1: Test cultural requirements

Type: System testing, non-functional, dynamic, manual

Initial state: N/A

Input: N/A

Output: No offensive images or text in source code or displayed to users

How the test will be performed: Manually test that there are no offensive images or text

NFR-C-2: Test cultural requirements

Type: System testing, non-functional, dynamic, manual

Initial state: N/A

Input: N/A

Output: Game console output is all in English
How the test will be performed: Manually test that everything displayed to the user is in English

3.2.7 Legal Requirements

NFR-L-1: Test legal requirements

Type: System testing, non-functional, static, manual
Initial state: N/A
Input: N/A
Output: Adherence to copyright properties
How the test will be performed: Manually test the adherence to copyright properties under the MIT license

3.3 Traceability Between Test Cases and Requirements

Table 4: Traceability Matrix: Functional Requirement

Test IDs	Requirement IDs							
	BE1	BE2	BE3	BE4	BE5	BE6	BE7	BE8
FR-C-1	X	X	X	X	X	X		
FR-C-2	X	X	X	X	X	X		
FR-C-3	X	X	X	X	X	X		
FR-C-4	X	X	X	X	X	X		
FR-C-5	X	X	X	X	X	X		
FR-C-6	X	X	X	X	X	X		
FR-C-7	X	X	X	X	X	X		
FR-C-8	X	X	X	X	X	X		
FR-DP-1			X			X		
FR-DP-2			X			X		
FR-SP-1		X						
FR-SP-2		X						
FR-CP-1		X						
FR-CP-2		X						
FR-CP-3		X						
FR-CP-4		X						
FR-CP-5		X						
FR-CP-6		X						
FR-CP-7		X						
FR-CP-8		X						
FR-H-1						X		
FR-H-2						X		
FR-H-3						X		
FR-H-4						X		
FR-H-5	X	X	X		X	X		
FR-H-6						X		
FR-H-7						X		
FR-H-8	X					X		
FR-H-9						X		

Test IDs	Requirement IDs							
	BE1	BE2	BE3	BE4	BE5	BE6	BE7	BE8
FR-P-1	X							
FR-P-2		X	X	X	X	X	X	
FR-P-3		X	X	X	X	X	X	
FR-P-4					X		X	
FR-P-5				X			X	
FR-P-6		X	X	X				
FR-P-7						X		
FR-P-8						X		
FR-P-9							X	
FR-P-10					X		X	
FR-P-11					X		X	
FR-P-12				X			X	
FR-P-13	X							
FR-P-14	X				X			
FR-P-15				X				
FR-P-16	X				X			
FR-M-1				X			X	
FR-M-2				X			X	
FR-M-3				X			X	
FR-M-4				X			X	
FR-M-5				X			X	
FR-M-6				X			X	
FR-M-7				X			X	
FR-UIO-1						X		
FR-UIO-21							X	
FR-UIO-32							X	
FR-UIO-43	X							
FR-UIO-54	X							
FR-UIO-65		X	X	X	X		X	
FR-UIO-76		X	X	X	X		X	
FR-UIO-87	X							

Test IDs	Requirement IDs							
	BE1	BE2	BE3	BE4	BE5	BE6	BE7	BE8
FR-GO-1							X	
FR-GO-2							X	
FR-GO-3							X	
FR-GO-4	X							X
FR-GO-5	X							X
FR-GO-6						X		
FR-GO-7						X		
FR-GO-86	X							
FR-GO-97	X							
FR-GO-10		X						
FR-GO-11			X					
FR-GO-128								X
FR-GO-139		X	X	X	X		X	
FR-GO-104	X							
FR-GO-115	X							
FR-GO-12		X						

Table 5: Traceability Matrix: Non-Functional Requirement

Test IDs	Requirement IDs								
	LF1	UH1	P1	P2	OE1	MS1	C1	C2	L1
NFR-LF-1	X								
NFR-UH-1		X							
NFR-P-1			X						
NFR-P-2				X					
NFR-OE-1					X				
NFR-MS-1						X			
NFR-C-1							X		
NFR-C-2								X	
NFR-L-1									X

4 System Tests for Proof of Concept

1. FC-1 (Testing Sequence Melds)

Type: Functional, dynamic, manual

Initial State: Hand of 10 cards with one sequence meld

Input/Condition: Hand of Cards

Output/Result: List of one sequence meld

How test will be performed: With a main method that calls on the public class

2. FC-2 (Testing Group Meld)

Type: Functional, dynamic, manual

Initial State: Hand of 10 cards with one group meld

Input/Condition: Hand of Cards

Output: List of one group melds

How test will be performed: With a main method, that calls on the public method

3. FC-3 (Testing drawing from discard pile)

Type: Functional, dynamic, manual

Initial State: Player has made a decision

Input: String input of the card the player wants to draw

Output: Card added to player hand

How test will be performed: Exploratory test of the whole game system.

4. FC-4 (Testing unique display of cards)

Type: Functional, dynamic, manual

Initial State: New Game

Input: N/A

Output: Game options

How test will be performed: Exploratory test of the whole game system.

5. FC-5 (Testing discard pile)

Type: Functional, dynamic, manual

Initial State: Player wants to check discard pile for card

Input: N/A

Output: Game options

How test will be performed: Exploratory test of the whole game system.

6. FC-6 (Testing computer turns)

Type: Functional, dynamic, manual

Initial State: New Game

Input: Scanner inputs

Output: Game options

How test will be performed: Exploratory test of the whole game system.

7. FC-7 (Testing drawing from stock pile

Type: Functional, dynamic, manual

Initial State: Player wants to draw from pile

Input: Scanner Input

Output: Card added to player hand

How test will be performed: Exploratory test of the whole game system.

8. FC-8 (Testing Check melds)

Type: Functional, dynamic, manual

Initial State: Player wants to check melds

Input: Scanner Input

Output: List of melds displayed

How test will be performed: Exploratory test of the whole game system.

9. FC-9 (Testing Check deadwood score/knock)

Type: Functional, dynamic, manual

Initial State: Players wants to check deadwood score or knock

Input: Scanner Input

Output: Deadwood score displayed and if applicable knock option prompted

How test will be performed: Exploratory test of the whole game system.

10. FC-10 (Testing discard a card)

Type: Functional, dynamic, manual

Initial State: Player wants to discard a card

Input: String representation of the card to be discarded.

Output: New hand with the previous card no longer there

How test will be performed: Exploratory test of the whole game system.

5 Comparison to Existing Implementation

These tests compare the program to the Go implementation.

- NFR-LF-1 Test Game Appearance
- NFR-MS-1 Test Maintainability Requirements

6 Unit Testing Plan

6.1 Unit testing of internal functions

Unit testing of internal functions will involve testing methods with return outputs. Assertion statements will be used to assess the correctness of the output from the internal functions. The expectation is that this output matches the expected output.

Coverage-based test cases will also be used. The goal is to ensure statement, condition and branch coverage. Boundary test-cases will be used for exception catching and handling, and to test user input.

6.2 Unit testing of output files

Not applicable for this system.

7 Appendix

N/A