Java test classes

DeckTest.java

```
package test;

import junit.framework.TestCase;
import main.Card;
import main.Deck;
import main.House;
import main.Rank;
import org.junit.Test;

/**
 * Units test for the class Deck
 */
public class DeckTest extends TestCase {

    //A valid array of cards
    private Card[] validDeckOfCardsArray = {
            new Card(House.SPADES, Rank.KING), new Card(House.SPADES,
Rank.QUEEN), new Card(House.SPADES, Rank.JACK), new Card(House.SPADES,
Rank.TEN),
            new Card(House.SPADES, Rank.NINE), new Card(House.SPADES,
Rank.EIGHT), new Card(House.SPADES, Rank.SEVEN), new Card(House.SPADES,
Rank.SIX),
            new Card(House.SPADES, Rank.FIVE), new Card(House.SPADES,
Rank.FOUR), new Card(House.SPADES, Rank.THREE), new Card(House.SPADES,
Rank.TWO),
            new Card(House.SPADES, Rank.ACE), new Card(House.DIAMONDS,
Rank.KING), new Card(House.DIAMONDS, Rank.QUEEN), new Card(House.DIAMONDS,
Rank.JACK),
            new Card(House.DIAMONDS, Rank.TEN), new Card(House.DIAMONDS,
Rank.NINE), new Card(House.DIAMONDS, Rank.EIGHT), new Card(House.DIAMONDS,
Rank.SEVEN),
            new Card(House.DIAMONDS, Rank.SIX), new Card(House.DIAMONDS,
Rank.FIVE), new Card(House.DIAMONDS, Rank.FOUR), new Card(House.DIAMONDS,
Rank.THREE),
            new Card(House.DIAMONDS, Rank.TWO), new Card(House.DIAMONDS,
Rank.ACE), new Card(House.CLUBS, Rank.KING), new Card(House.CLUBS,
Rank.QUEEN),
            new Card(House.CLUBS, Rank.JACK), new Card(House.CLUBS,
Rank.TEN), new Card(House.CLUBS, Rank.NINE), new Card(House.CLUBS,
Rank.EIGHT),
            new Card(House.CLUBS, Rank.SEVEN), new Card(House.CLUBS,
Rank.SIX), new Card(House.CLUBS, Rank.FIVE), new Card(House.CLUBS,
Rank.FOUR),
            new Card(House.CLUBS, Rank.THREE), new Card(House.CLUBS,
Rank.TWO), new Card(House.CLUBS, Rank.ACE), new Card(House.HEARTS,
Rank.KING),
            new Card(House.HEARTS, Rank.QUEEN), new Card(House.HEARTS,
Rank.JACK), new Card(House.HEARTS, Rank.TEN), new Card(House.HEARTS,
Rank.NINE),
            new Card(House.HEARTS, Rank.EIGHT), new Card(House.HEARTS,
Rank.SEVEN), new Card(House.HEARTS, Rank.SIX), new Card(House.HEARTS,
Rank.FIVE),
            new Card(House.HEARTS, Rank.FOUR), new Card(House.HEARTS,
Rank.THREE), new Card(House.HEARTS, Rank.TWO), new Card(House.HEARTS,
```

```
Rank.ACE)
    };

    /**
     * Utility method used in this test file to check if a card is within a
Card Array, as we cant use Java.Arrays Lib.
     *
     * @param card card to search for.
     * @return true if the input card can be found in the valid array of
card
     */
    private boolean validArrayOfCardsContains(Card card) {
        boolean found = false;
        for (int i = 0; i < validDeckOfCardsArray.length; i++) {
            if
(card.toString().equals(validDeckOfCardsArray[i].toString())) {
                found = true;
                break;
            }
        }
        return found;
    }

    /**
     * Test that create fullDeckOfCards creates a valid deck of cards.\
     */
    @Test
    public void testCreateFullDeckOfCards() {
        //Setup test
        Deck actualDeck = new Deck();
        Deck expectedDeck = new Deck();

        //Actual
        actualDeck.createFullDeckOfCards();

        //for each house, loop through each rank and push that card to the
stack.
        for (Card card : validDeckOfCardsArray) {
            expectedDeck.push(card);
        }

        Card[] actualAsArray = actualDeck.toArray();
        Card[] expectedAsArray = expectedDeck.toArray();

        // We had to do this as we could not use the library Arrays.
        // this was the only successful solution we could find without
using illegal libraries, as we are not allowed to use an java collections.
        boolean testResult = true;
        for (int i = 0; i < actualAsArray.length; i++) {
            if (actualAsArray[i].getHouse() !=
expectedAsArray[i].getHouse() && actualAsArray[i].getRank() !=
expectedAsArray[i].getRank()) {
                testResult = false;
            }
        }

        //Assert
        assertTrue(testResult);
    }

    /**
```

```
     * 1. Test push() can push a card to an empty deck.
     * 2. Test push() can push a card to a stack with a top Card already
existing.
     */
    @Test
    public void testPush() {
        //Setup test
        Deck testDeck = new Deck();

        testDeck.push(new Card(House.SPADES, Rank.ACE));

        //Assertions
        //1. Test push() can push a card to an empty deck..
        assertEquals("Ace of Spades", testDeck.peek().toString());

        testDeck.push(new Card(House.SPADES, Rank.TWO));

        //2. Test push() can push a card to a stack with a top Card already
existing.
        assertEquals("2 of Spades", testDeck.peek().toString());
    }

    /**
     * 1.Test that pop() removes a card and returns it from the top of the
deck.
     * 2.Test that pop() returns null when there is no top card.
     */
    @Test
    public void testPop() {
        //Setup test
        Deck testDeck = new Deck();
        testDeck.push(new Card(House.SPADES, Rank.ACE));

        var actual = testDeck.pop();
        var expected = new Card(House.SPADES, Rank.ACE);

        //1.Test that pop() removes a card and returns it from the top of
the deck.
        assertEquals(actual.toString(), expected.toString());

        //2.Test that pop() returns null when there is no top card.
        assertNull(testDeck.peek());
    }

    /**
     * Test to peek the top card of a deck, should return the top card of
the deck, and not remove it
     */
    @Test
    public void testPeek() {
        //Setup test
        Deck testDeck = new Deck();
        testDeck.push(new Card(House.SPADES, Rank.ACE));

        var testPeek = testDeck.peek();

        //Assert peek top card
        assertEquals("Ace of Spades", testPeek.toString());

        //Second Assert to make sure the last peek did not remove the Card.
        assertEquals("Ace of Spades", testPeek.toString());
```

```
    }

    /**
     * Test to check if isEmpty returns true when a deck is empty
     */
    @Test
    public void testIsEmpty() {
        //Setup test
        Deck testDeck = new Deck();

        //Assert
        assertTrue(testDeck.isEmpty());
    }

    /**
     * Make sure .clear() clears the deck
     */
    @Test
    public void testClear() {
        //Setup test
        Deck testDeck = new Deck();

        testDeck.push(new Card(House.SPADES, Rank.ACE));

        testDeck.clear();

        //Assert
        assertNull(testDeck.peek());
    }

    /**
     * Test rigorousShuffle() to make sure the deck is not altered, eg a
card is remove and the deck is no longer
     * a valid 52 card deck.
     */
    @Test
    public void testRigorousShuffle() {
        Deck actualDeck = new Deck();
        actualDeck.createFullDeckOfCards();

        //shuffle
        actualDeck.rigorousShuffle();

        //make sure shuffle does alter the deck.
        Card[] actualAsArray = actualDeck.toArray();

        // We had to do this as we could not use the library Arrays.
        boolean testResult = true;
        for (int i = 0; i < actualAsArray.length; i++) {
            if (!validArrayOfCardsContains(actualAsArray[i])) {
                //set to false if we cant find the card.
                testResult = false;
            }
        }

        //Assert
        assertTrue(testResult);
    }
}
```

GameMechanicsTest.java

```
package test;

import main.Card;
import main.GameMechanics;
import main.House;
import main.Rank;
import org.junit.Assert;
import org.junit.Test;

import static org.junit.Assert.*;

/**
 * Unit tests for GameMechanics Class
 */
public class GameMechanicsTest {

    private Card testCardAce  = new Card(House.CLUBS, Rank.ACE);
    private Card testCardTen  = new Card(House.CLUBS, Rank.TEN);
    private Card testCardKing = new Card(House.CLUBS, Rank.KING);
    private Card testCardQueen = new Card(House.CLUBS, Rank.QUEEN);
    private Card testCardJack = new Card(House.CLUBS, Rank.JACK);

    /**
     * Unit test for isFaceCard
     *
     * Test 1, returns true when the card param is a face Card.
     * Test 2, returns false when the card param is a not a face Card.
     */
    @Test
    public void isFaceCard() {
        //Test 1, returns true when the card param is a face Card.
        Assert.assertTrue(GameMechanics.isFaceCard(testCardKing));

        //Test 2, returns false when the card param is a not a face Card.
        Assert.assertFalse(GameMechanics.isFaceCard(testCardAce));
    }

    /**
     * Unit Test for isFacePairs
     *
     * Test 1, returns true when the cards supplied are face pairs eg King,
Queen and Jack.
     * Test 2, returns false when the cards supplied are not face pairs.
     * Test 3, returns false when the cards supplied are not face pairs and
one is null.
     */
    @Test
    public void isFacePairs() {
        //Test 1, returns true when the cards supplied are face pairs eg
King, Queen and Jack.
        Assert.assertTrue(GameMechanics.isFacePairs(testCardKing,
testCardJack, testCardQueen));

        //Test 2, returns false when the cards supplied are not face pairs.
        Assert.assertFalse(GameMechanics.isFacePairs(testCardAce,
testCardJack, testCardQueen));

        //Test 3, returns false when the cards supplied are not face pairs
and one is null.
```

```
        Assert.assertFalse(GameMechanics.isFacePairs(testCardAce,
testCardJack, null));
    }

    /**
     * Unit tests for isElevensPair
     *
     * Test 1, returns true when isElevensPair() input params are an
elevens pair
     * Test 2, returns false when isElevensPair() input params are not an
elevens pair
     * Test 3, returns false when isElevensPair() input params are not an
elevens pair and 1 is null
     */
    @Test
    public void isElevensPair() {
        //Test 1, returns true when isElevensPair() input params are an
elevens pair
        Assert.assertTrue(GameMechanics.isElevensPair(testCardAce,
testCardTen));

        //Test 2, returns false when isElevensPair() input params are not
an elevens pair
        Assert.assertFalse(GameMechanics.isElevensPair(testCardAce,
testCardJack));

        //Test 3, returns false when isElevensPair() input params are not
an elevens pair and 1 is null
        Assert.assertFalse(GameMechanics.isElevensPair(testCardAce, null));
    }

    /**
     * Units tests for cardSelectionCharToInt
     *
     * Test 1, GameMechanics.cardSelectionCharToInt() returns 0 when input
is 'a'
     * Test 2, GameMechanics.cardSelectionCharToInt() returns 1 when input
is 'b'
     * Test 3, GameMechanics.cardSelectionCharToInt() returns 2 when input
is 'c'
     * Test 4, GameMechanics.cardSelectionCharToInt() returns 3 when input
is 'd'
     * Test 5, GameMechanics.cardSelectionCharToInt() returns 4 when input
is 'e'
     * Test 6, GameMechanics.cardSelectionCharToInt() returns 5 when input
is 'f'
     * Test 7, GameMechanics.cardSelectionCharToInt() returns 6 when input
is 'g'
     * Test 8, GameMechanics.cardSelectionCharToInt() returns 7 when input
is 'h'
     * Test 9, GameMechanics.cardSelectionCharToInt() returns 8 when input
is 'i'
     * Test 10, GameMechanics.cardSelectionCharToInt() returns -1 when
input is 'z'
     */
    @Test
    public void cardSelectionCharToInt() {
        //Test 1, GameMechanics.cardSelectionCharToInt() returns 0 when
input is 'a'
        Assert.assertEquals(0, GameMechanics.cardSelectionCharToInt('a'));
```

```
        //Test 2, GameMechanics.cardSelectionCharToInt() returns 1 when
input is 'b'
        Assert.assertEquals(1, GameMechanics.cardSelectionCharToInt('b'));

        //Test 3, GameMechanics.cardSelectionCharToInt() returns 2 when
input is 'c'
        Assert.assertEquals(2, GameMechanics.cardSelectionCharToInt('c'));

        //Test 4, GameMechanics.cardSelectionCharToInt() returns 3 when
input is 'd'
        Assert.assertEquals(3, GameMechanics.cardSelectionCharToInt('d'));

        //Test 5, GameMechanics.cardSelectionCharToInt() returns 4 when
input is 'e'
        Assert.assertEquals(4, GameMechanics.cardSelectionCharToInt('e'));

        //Test 6, GameMechanics.cardSelectionCharToInt() returns 5 when
input is 'f'
        Assert.assertEquals(5, GameMechanics.cardSelectionCharToInt('f'));

        //Test 7, GameMechanics.cardSelectionCharToInt() returns 6 when
input is 'g'
        Assert.assertEquals(6, GameMechanics.cardSelectionCharToInt('g'));

        //Test 8, GameMechanics.cardSelectionCharToInt() returns 7 when
input is 'h'
        Assert.assertEquals(7, GameMechanics.cardSelectionCharToInt('h'));

        //Test 9, GameMechanics.cardSelectionCharToInt() returns 8 when
input is 'i'
        Assert.assertEquals(8, GameMechanics.cardSelectionCharToInt('i'));

        //Test 10, GameMechanics.cardSelectionCharToInt() returns -1 when
input is 'z'
        Assert.assertEquals(-1, GameMechanics.cardSelectionCharToInt('z'));
    }


    /**
     * Unit tests for cardSelectionNumberToString
     *
     * Test 1, GameMechanics.cardSelectionCharToInt() returns 'a' when
input is 0
     * Test 2, GameMechanics.cardSelectionCharToInt() returns 'b' when
input is 1
     * Test 3, GameMechanics.cardSelectionCharToInt() returns 'c' when
input is 2
     * Test 4, GameMechanics.cardSelectionCharToInt() returns 'd' when
input is 3
     * Test 5, GameMechanics.cardSelectionCharToInt() returns 'e' when
input is 4
     * Test 6, GameMechanics.cardSelectionCharToInt() returns 'f' when
input is 5
     * Test 7, GameMechanics.cardSelectionCharToInt() returns 'g' when
input is 6
     * Test 8, GameMechanics.cardSelectionCharToInt() returns 'h' when
input is 7
     * Test 9, GameMechanics.cardSelectionCharToInt() returns 'i' when
input is 8
     * Test 10, GameMechanics.cardSelectionCharToInt() returns -'z' when
input is -1
```

```
     */
    @Test
    public void cardSelectionNumberToString() {
        //Test 1, GameMechanics.cardSelectionNumberToString() returns a
when input is 0
        Assert.assertEquals("a",
GameMechanics.cardSelectionNumberToString(0));

        //Test 2, GameMechanics.cardSelectionNumberToString() returns b
when input is 1
        Assert.assertEquals("b",
GameMechanics.cardSelectionNumberToString(1));

        //Test 3, GameMechanics.cardSelectionNumberToString() returns c
when input is 2
        Assert.assertEquals("c",
GameMechanics.cardSelectionNumberToString(2));

        //Test 4, GameMechanics.cardSelectionNumberToString() returns d
when input is 3
        Assert.assertEquals("d",
GameMechanics.cardSelectionNumberToString(3));

        //Test 5, GameMechanics.cardSelectionNumberToString() returns e
when input is 4
        Assert.assertEquals("e",
GameMechanics.cardSelectionNumberToString(4));

        //Test 6, GameMechanics.cardSelectionNumberToString() returns f
when input is 5
        Assert.assertEquals("f",
GameMechanics.cardSelectionNumberToString(5));

        //Test 7, GameMechanics.cardSelectionNumberToString() returns g
when input is 6
        Assert.assertEquals("g",
GameMechanics.cardSelectionNumberToString(6));

        //Test 8, GameMechanics.cardSelectionNumberToString() returns h
when input is 7
        Assert.assertEquals("h",
GameMechanics.cardSelectionNumberToString(7));

        //Test 9, GameMechanics.cardSelectionNumberToString() returns i
when input is 8
        Assert.assertEquals("i",
GameMechanics.cardSelectionNumberToString(8));

        //Test 10, GameMechanics.cardSelectionNumberToString() returns
ERROR when input is 99
        Assert.assertEquals("ERROR",
GameMechanics.cardSelectionNumberToString(99));
    }

    /**
     * Unit tests for validStringSelection
     *
     * Test 1, returns true when 3 valid chars are selected
     * Test 2, returns true when 2 valid chars are selected
     * Test 3, returns false when 3 invalid chars are selected
     * Test 4, returns false when 2 invalid chars are selected
```

```
     * Test 5, returns false when 2 valid and 1 invalid chars are selected
     * Test 6, returns false when 1 valid and 1 invalid chars are selected
     * Test 7, returns false when 1 letter is provided
     * Test 8, returns false when 0 letters is provided
     */
    @Test
    public void validStringSelection() {
        // Test 1, returns true when 3 valid chars are selected
        Assert.assertTrue(GameMechanics.validStringSelection("abc"));

        // Test 2, returns true when 2 valid chars are selected
        Assert.assertTrue(GameMechanics.validStringSelection("de"));

        // Test 3, returns false when 3 invalid chars are selected
        Assert.assertFalse(GameMechanics.validStringSelection("xyz"));

        // Test 4, returns false when 2 invalid chars are selected
        Assert.assertFalse(GameMechanics.validStringSelection("pq"));

        // Test 5, returns false when 2 valid and 1 invalid chars are
selected
        Assert.assertFalse(GameMechanics.validStringSelection("abz"));

        // Test 6, returns false when 1 valid and 1 invalid chars are
selected
        Assert.assertFalse(GameMechanics.validStringSelection("az"));

        // Test 7, returns false when 1 letter is provided
        Assert.assertFalse(GameMechanics.validStringSelection("a"));

        // Test 8, returns false when 0 letters is provided
        Assert.assertFalse(GameMechanics.validStringSelection("a"));
    }

    /**
     * Unit tests for allowedCharacter
     *
     * Test 1, returns true when input is a
     * Test 2, returns true when input is b
     * Test 3, returns true when input is c
     * Test 4, returns true when input is d
     * Test 5, returns true when input is e
     * Test 6, returns true when input is f
     * Test 7, returns true when input is g
     * Test 8, returns true when input is h
     * Test 9, returns true when input is i
     * Test 10, returns false when not one of (a b c d e f g h i)
     */
    @Test
    public void allowedCharacter() {
        // Test 1, returns true when input is a
        Assert.assertTrue(GameMechanics.allowedCharacter('a'));

        // Test 2, returns true when input is b
        Assert.assertTrue(GameMechanics.allowedCharacter('b'));

        // Test 3, returns true when input is c
        Assert.assertTrue(GameMechanics.allowedCharacter('c'));

        // Test 4, returns true when input is d
        Assert.assertTrue(GameMechanics.allowedCharacter('d'));
```

```
        // Test 5, returns true when input is e
        Assert.assertTrue(GameMechanics.allowedCharacter('e'));

        // Test 6, returns true when input is f
        Assert.assertTrue(GameMechanics.allowedCharacter('f'));

        // Test 7, returns true when input is g
        Assert.assertTrue(GameMechanics.allowedCharacter('g'));

        // Test 8, returns true when input is h
        Assert.assertTrue(GameMechanics.allowedCharacter('h'));

        // Test 9, returns true when input is i
        Assert.assertTrue(GameMechanics.allowedCharacter('i'));

        // Test 10, returns false when not one of (a b c d e f g h i)
        Assert.assertFalse(GameMechanics.allowedCharacter('z'));
    }
}
```

GameTest.java

```
package test;

import main.Game;
import org.junit.Assert;
import org.junit.Test;

import static org.junit.Assert.*;

/**
 * Unit tests for Game Class
 *
 * Developer Note: unit tests here are bare as we do not have many Setters
for this class.
 * As they are not required for this object.
 *
 * Methods that have only be tested Manually, as we cannot via unit test
without illegal libraries.
 * computerDemonstrationGame()
 * userPlayableGame()
 * getDeck()
 * getDiscardDeck()
 */
public class GameTest {

    /**
     * Test 1 getRoundQueue should return null, when a new game is created
as no round Queue has been created
     */
    @Test
    public void getRoundQueue() {
        Game game = new Game();

        //Test 1 getRoundQueue should return null, when a new game is
created as no rounds have be generated
        assertNull(game.getRoundQueue());
    }
```

```
    /**
     * Test 1 getCurrentRound should return null, when a new game is
created as no rounds has been created
     */
    @Test
    public void getCurrentRound() {
        Game game = new Game();

        //Test 1 getCurrentRound should return null, when a new game is
created as no rounds have be generated
        assertNull(game.getCurrentRound());
    }

    /**
     * Test 1 .getGameResult() returns false when a unplayed game object is
created.
     */
    @Test
    public void getGameResult() {
        Game game = new Game();

        //Test 1 .getGameResult() returns false when a unplayed game object
is created.
        Assert.assertFalse(game.getGameResult());
    }
}
```

HouseTest.java

```
package test;

import junit.framework.TestCase;
import main.House;

/**
 * Units Tests for the Class House
 */
public class HouseTest extends TestCase {

    /**
     * Test for House.toString()
     */
    public void testToString() {
        //Test House.CLUBS.toString() equals "Clubs"
        assertEquals(House.CLUBS.toString(), "Clubs");

        //Test House.DIAMONDS.toString() equals "Diamonds"
        assertEquals(House.DIAMONDS.toString(), "Diamonds");

        //Test House.SPADES.toString() equals "Spades"
        assertEquals(House.SPADES.toString(), "Spades");

        //Test House.HEARTS.toString() equals "Hearts"
        assertEquals(House.HEARTS.toString(), "Hearts");
    }
}
```

RankTest.java

```java
package test;

import junit.framework.TestCase;
import main.Rank;

/**
 * Unit tests for the class Rank
 */
public class RankTest extends TestCase {

    private Rank testRankKing = Rank.KING;
    private Rank testRankAce = Rank.ACE;
    private Rank testRankTwo = Rank.TWO;

    /**
     * Test getRank() returns the correct Rank of the Card
     */
    public void testGetRank() {
        //a card with Rank.KING .getRank() should equal "King"
        assertEquals(testRankKing.getRank(), "King");

        //a card with Rank.ACE .getRank() should equal "Ace"
        assertEquals(testRankAce.getRank(), "Ace");

        //a card with Rank.TWO .getRank() should equal "Two"
        assertEquals(testRankTwo.getRank(), "Two");
    }

    /**
     * Test getValue() returns the correct value of the card
     */
    public void testGetValue() {
        //a card with Rank.KING .getRank() should equal -1
        assertEquals(testRankKing.getValue(), -1);

        //a card with Rank.Ace .getRank() should equal 1
        assertEquals(testRankAce.getValue(), 1);

        //a card with Rank.Two .getRank() should equal 2
        assertEquals(testRankTwo.getValue(), 2);
    }

    /**
     * Test toString returns the Rank as a String
     */
    public void testTestToString() {
        //a card with Rank.KING .toString() should equal "King"
        assertEquals(testRankKing.toString(), "King");

        //a card with Rank.ACE .toString() should equal "Ace"
        assertEquals(testRankAce.toString(), "Ace");

        //a card with Rank.TWO .toString() should equal "Two"
        assertEquals(testRankTwo.toString(), "Two");
    }
}
```

RoundQueueTest.java

```
package test;

import main.CardSlotsBag;
import main.Round;
import main.RoundQueue;
import org.junit.Assert;
import org.junit.Test;

public class RoundQueueTest {

    /**
     * Test RoundQueue enqueue() method
     */
    @Test
    public void enqueue() {

        Round front = new Round(0, new CardSlotsBag());
        RoundQueue roundQueue = new RoundQueue();

        /*
         * Should add Round to the front of the queue
         */

        //Assert is currently null
        Assert.assertNull(roundQueue.getFront());

        roundQueue.enqueue(front);

        var actual = roundQueue.getFront();
        var expected = front;

        //Assert .enqueue(front) added the round.
        Assert.assertEquals(actual, expected);


        /*
         * Should add Round too be the next round after the first Round and
be the rear Node.
         */
        Round nextRound = new Round(0, new CardSlotsBag());
        roundQueue.enqueue(nextRound);

        var actual2 = roundQueue.getRear();
        var expected2 = nextRound;

        //Assert that is nextRound is rear
        Assert.assertEquals(actual2, expected2);

    }

    /**
     * Test RoundQueue dequeue() method
     */
    @Test
    public void dequeue() {
        Round front = new Round(0, new CardSlotsBag());
        Round rear = new Round(1, new CardSlotsBag());
        RoundQueue roundQueue = new RoundQueue();
        /*
```

```
         * Should not remove a round from an empty queue
         */
        var actual = roundQueue.getFront();

        //Assert roundQueue is null
        Assert.assertNull(actual);

        //Assert roundQueue is still null after a .dequeue null
        roundQueue.dequeue();
        Assert.assertNull(actual);


        /*
         * Should remove a round from a queue with 1 Round in it.
         */
        //Assert queue is not null
        roundQueue.enqueue(front);
        Assert.assertNotNull(roundQueue.getFront());

        //Assert queue is null after calling .dequeue()
        roundQueue.dequeue();
        var actual1 = roundQueue.getFront();
        Assert.assertNull(actual1);


        /*
         * Should remove a round from a queue with 2 Round's in it.
         * and the rear round should now be both the front and rear
         */
        roundQueue.enqueue(front);
        roundQueue.enqueue(rear);

        //Assert front and rear are correct
        Assert.assertEquals(roundQueue.getFront(), front);
        Assert.assertEquals(roundQueue.getRear(), rear);

        //Assert dequeue removes the front round in the que, so that rear
is now rear and front
        roundQueue.dequeue();
        Assert.assertEquals(roundQueue.getFront(), rear);
        Assert.assertEquals(roundQueue.getRear(), rear);
    }

    /**
     * Test RoundQueue getFront method
     */
    @Test
    public void getFront() {
        Round front = new Round(0, new CardSlotsBag());
        RoundQueue roundQueue = new RoundQueue();

        /*
          Test getFront() returns the front Round.
         */
        roundQueue.enqueue(front);

        var expected = front;
        var actual = roundQueue.getFront();

        Assert.assertEquals(expected, actual);
    }
```

```
    /**
     * Test RoundQueues isEmpty Method
     */
    @Test
    public void isEmpty() {
        Round front = new Round(0, new CardSlotsBag());
        RoundQueue roundQueue = new RoundQueue();
        /*
          Test isEmpty returns true when the RoundQueue is empty.
         */
        var expected = true;
        var actual = roundQueue.isEmpty();

        Assert.assertEquals(expected, actual);

        /*
          Test isEmpty returns false when the RoundQueue is not empty.
         */
        roundQueue.enqueue(front);
        var expected1 = false;
        var actual1 = roundQueue.isEmpty();

        Assert.assertEquals(expected1, actual1);
    }

    /**
     * Test RoundQueues clear() method
     */
    @Test
    public void clear() {
        Round front = new Round(0, new CardSlotsBag());
        RoundQueue roundQueue = new RoundQueue();

        /*
          Test clear clears the RoundQueue, when it was not empty.
         */

        //first add and element and assert is not empty
        roundQueue.enqueue(front);
        Assert.assertEquals(roundQueue.getFront(), front);

        //Assert clear clears the queue and front and rear equal null
        roundQueue.clear();
        Assert.assertNull(roundQueue.getFront());
        Assert.assertNull(roundQueue.getRear());
    }
}
```

RoundTest.java

```
package test;

import junit.framework.TestCase;
import main.*;

/**
 * Unit tests for the Round Class methods.
 */
public class RoundTest extends TestCase {

    /**
     * Test 1 Returns true when there is no elevens pairs and no pairs of
face cards
     * Test 2 Returns false when there is a elevens pair
     * Test 3 Returns false when there is face card pairs.
     */
    public void testIsStalemate() {
        //Test 1 Returns true when there is no elevens pairs and no pairs
of face cards
        CardSlotsBag bag1 = new CardSlotsBag();
        bag1.addNewEntry(new Card(House.DIAMONDS, Rank.ACE));
        bag1.addNewEntry(new Card(House.CLUBS, Rank.ACE));
        Round round1 = new Round(1, bag1);
        //Assert Test 1
        assertTrue(round1.isStalemate());

        //Test 2 Returns false when there is a elevens pair
        CardSlotsBag bag2 = new CardSlotsBag();
        bag2.addNewEntry(new Card(House.DIAMONDS, Rank.ACE));
        bag2.addNewEntry(new Card(House.DIAMONDS, Rank.TEN));
        Round round2 = new Round(2, bag2);
        //Assert Test 2
        assertFalse(round2.isStalemate());

        //Test 3Returns false when there is face card pairs.
        CardSlotsBag bag3 = new CardSlotsBag();
        bag3.addNewEntry(new Card(House.DIAMONDS, Rank.ACE));
        bag3.addNewEntry(new Card(House.DIAMONDS, Rank.TEN));
        bag3.addNewEntry(new Card(House.CLUBS, Rank.KING));
        bag3.addNewEntry(new Card(House.DIAMONDS, Rank.QUEEN));
        bag3.addNewEntry(new Card(House.CLUBS, Rank.JACK));
        Round round3 = new Round(2, bag2);

        //Assert Test 3
        assertFalse(round3.isStalemate());
    }

    /**
     * Test 1
     * replaceEmptyCardSlots() should add 9 cards to bag1
     * Test 2
     * replaceEmptyCardSlots should add 1 card to the bag2 when bag2 has 8
cards inside it.
     */
    public void testReplaceEmptyCardSlots() {
        Deck deck = new Deck();
        deck.createFullDeckOfCards();

        // Test 1
```

```
        // replaceEmptyCardSlots() should add 9 cards to bag1
        CardSlotsBag bag1 = new CardSlotsBag();
        Round round1 = new Round(1, bag1);
        assertEquals(bag1.countCards(), 0);
        round1.replaceEmptyCardSlots(deck);

        //Assert Test 1
        assertEquals(bag1.countCards(), 9);

        // Test 2
        // ReplaceEmptyCardSlots should add 1 card to the bag2 when bag2
has 8 cards inside it.
        CardSlotsBag bag2 = new CardSlotsBag();
        bag2.addNewEntry(new Card(House.CLUBS, Rank.ACE));
        bag2.addNewEntry(new Card(House.CLUBS, Rank.TWO));
        bag2.addNewEntry(new Card(House.CLUBS, Rank.THREE));
        bag2.addNewEntry(new Card(House.CLUBS, Rank.FOUR));
        bag2.addNewEntry(new Card(House.CLUBS, Rank.FIVE));
        bag2.addNewEntry(new Card(House.CLUBS, Rank.SIX));
        bag2.addNewEntry(new Card(House.CLUBS, Rank.SEVEN));
        bag2.addNewEntry(new Card(House.CLUBS, Rank.EIGHT));
        Round round2 = new Round(2, bag2);

        //Assertions Test 2
        assertEquals(bag2.countCards(), 8);
        round2.replaceEmptyCardSlots(deck);
        assertEquals(bag2.countCards(), 9);
    }

    /**
     * getRoundNumber() should return 1
     */
    public void testGetRoundNumber() {
        CardSlotsBag bag1 = new CardSlotsBag();
        Round round1 = new Round(1, bag1);
        assertEquals(round1.getRoundNumber(), 1);
    }

    /**
     * setRoundNumber() should set the round number to 2.
     */
    public void testSetRoundNumber() {
        CardSlotsBag bag1 = new CardSlotsBag();
        Round round1 = new Round(1, bag1);
        round1.setRoundNumber(2);
        assertEquals(round1.getRoundNumber(), 2);
    }

    /**
     * getCardSlotBag should return a empty bag.
     */
    public void testGetCardSlotBag() {
        CardSlotsBag bag1 = new CardSlotsBag();
        Round round1 = new Round(1, bag1);
        assertEquals(round1.getCardsInPlayBag(), bag1);
    }

    /**
     * setCardSlotBag should set the rounds bag 'bagWithCard' with a Card
in it
     */
```

```
    public void testSetCardSlotBag() {
        CardSlotsBag bag1 = new CardSlotsBag();
        CardSlotsBag bagWithCard = new CardSlotsBag();
        Round round1 = new Round(1, bag1);
        bagWithCard.addNewEntry(new Card(House.CLUBS, Rank.ACE));
        round1.setCardsInPlayBag(bagWithCard);
        assertEquals(round1.getCardsInPlayBag(), bagWithCard);
    }

    /**
     * round1.getNextRound() should return  CardSlotsBag round2
     */
    public void testGetNextRound() {
        CardSlotsBag bag1 = new CardSlotsBag();
        CardSlotsBag bag2 = new CardSlotsBag();
        Round round1 = new Round(1, bag1);
        Round round2 = new Round(2, bag2);
        round1.setNextRound(round2);
        assertEquals(round1.getNextRound(), round2);
    }

    /**
     * round1.setNext Round should set round1's nextRound to Round2.
     */
    public void testSetNextRound() {
        CardSlotsBag bag1 = new CardSlotsBag();
        CardSlotsBag bag2 = new CardSlotsBag();
        Round round1 = new Round(1, bag1);
        Round round2 = new Round(2, bag2);
        round1.setNextRound(round2);
        assertEquals(round1.getNextRound(), round2);
    }
}
```

ElevensTest.java

```
package test;

/**
 * Developer Note: Testing is not required here as to test we require
illegal libraries
 * due to System.in requirements.
 *
 * So test for this class we be performed manually
 *
 * This class will not be run when unit tests are ran.
 */
public class ElevensTest {

}
```

ColorsTest.java

```java
package test;

/**
 * Developer Note: Testing for the Class is not required.
 *
 * This class will not be run when unit tests are ran.
 */
public class ColorsTest {}
```

DisplayTest.java

```java
package test;

/**
 * Developer Note: All Testing for Display will be perform manually as all
methods here only
 * perform System.out
 *
 * This class will not be run when unit tests are ran.
 */
public class DisplayTest {}
```