ECM2414 Pair Programming Coursework

Thomas Newbold – 71000126

Steven Jangcan – 710042102

Weighting 50:50

Contents:

Development Log

Date	Time	Duration (hh:mm)	Roles	
			71000126	710042102
08/11/2022	11:30	01:00	Observer	Driver
08/11/2022	12:30	01:00	Driver	Observer
12/11/2022	10:00	01:30	Driver	Observer
12/11/2022	11:30	01:30	Observer	Driver
12/11/2022	01:00	01:00	Driver	Observer
12/11/2022	02:00	01:00	Observer	Driver
15/11/2022	11:30	01:00	Observer	Driver
15/11/2022	12:30	01:00	Driver	Observer
15/11/2022	02:30	00:30	Observer	Driver
18/11/2022	01:30	01:00	Driver	Observer
18/11/2022	02:30	01:00	Observer	Driver
18/11/2022	03:30	00:30	Driver	Observer
20/11/2022	04:30	01:00	Driver	Observer
22/11/2022	11:30	01:00	Observer	Driver
22/11/2022	12:30	01:00	Driver	Observer
22/11/2022	02:30	01:00	Driver	Observer

Design

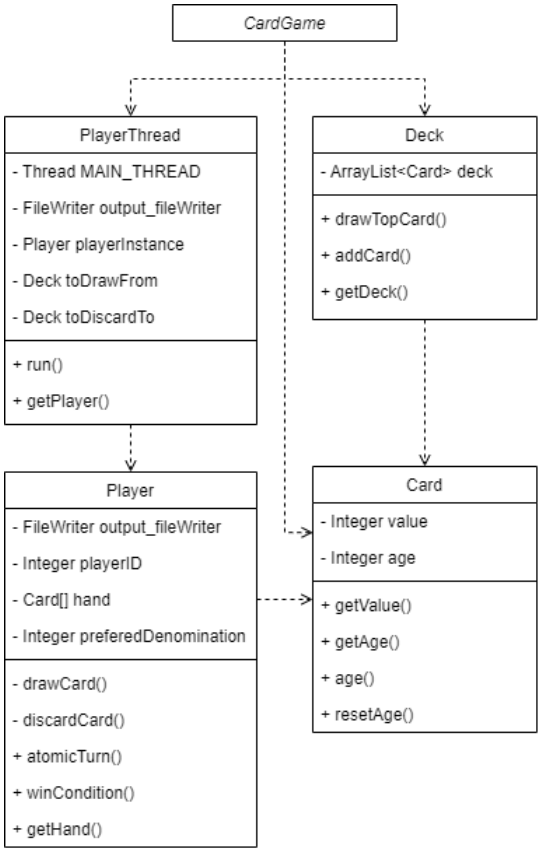

Figure 1: UML Class Diagram

The overall (high-level) design for the system can be seen in this UML diagram *(Fig. 1, left)*. Most classes are used by the CardGame class, within its main loop.

```java
public class CardGame {
    public static Integer players;
    public static String packFile;
    public volatile static Integer winPlayer;

    /* MAINLOOP:
     * Takes input and validates
     *       - positive integer for player count
     *       - valid pack file (8n cards; contains at least one winning hand)
     * Deals cards to players and decks (each containing 4 cards)
     * Creates player threads
     * Initial win condition check; break if met
     * Start player threads; halt main thread
     * When Player wins and notifys main thread, interrupt remaining threads
     * Write output files for players/decks
     */
    Run | Debug
    public static void main(String[] args) {}
}
```

Figure 2: CardGame class outline

The main CardGame class *(Fig. 2, shown above)* deals with input validation, dealing the cards, player thread creation, and initialising the associated output files.

Provided below are the outlines for supporting classes, with docstrings describing each function to be implemented. Instance variables have been confined by using private variables and getter methods.

```java
import java.util.ArrayList;

public class Deck {
    private ArrayList<Card> deck = new ArrayList<Card>();
    public Deck(Card[] cards) {}

    /**
     * Draws the top card from the deck
     * @return The drawn card
     */
    public synchronized Card drawTopCard() { return new Card(0); }

    /**
     * Adds a card to the bottom of the deck
     * @param c The card to be added to the deck
     */
    public synchronized void addCard(Card c) {}

    /**
     * @return The deck as an array of Cards
     */
    public Card[] getDeck() { return new Card[0]; }
}
```

Figure 3: Deck class outline

```java
public class Card {
    private Integer value;
    private Integer age;
    public Card(Integer v) {}

    /**
     * Only getter method; value is read-only so no syncronisation required
     * @return The value of the card
     */
    public Integer getValue() { return 0; }

    /**
     * @return The "age" of the card (number of turns it has been in the players hand)
     */
    public Integer getAge() { return 0; }
    /**
     * "Ages" the card; used to prevent stale hands
     */
    public void age() {}
    /**
     * Sets the age to zero (when a card is picked up from the deck)
     */
    public void resetAge() {}
}
```

Figure 4: Card class outline

The Deck class *(Fig. 3, above, left)* has been synchronised, as it is likely that different player threads will be adding/removing cards simultaneously. This is the only data structure that will be accessed by multiple threads (at the same time; instances of Card will be passed between threads but never modified during this time). The Card class *(Fig. 4, above, right)* simply holds a value, and an "age" (which increments each turn a card spends in a player's hand) – this value is used when choosing a card to discard, and prevents stale hands.

```java
import java.util.ArrayList;
import java.io.FileWriter;

public class Player {
    private FileWriter fw;
    private Integer playerID;
    private ArrayList<Card> hand = new ArrayList<Card>();
    private Integer preferredDenom;
    public Player(FileWriter fileWriter, Integer pID, Card[] cards) {}

    /**
     * Draws card from the referenced deck and adds to Players hand
     * @param d The deck to draw from
     */
    private void drawCard(Deck d) {}

    /**
     * Chooses and discards a card from the Player's hand (not of prefered denomination)
     * @returns The discarded card
     */
    private Card discardCard() { return new Card[0]; }

    /**
     * Performs a players full turn atomically (before checking win condition);
     * Player's hand will only ever contain 4 cards outside of this function;
     * Writes drawn and discarded cards, as well as hand, to the output file
     * @param d1 The deck to draw from
     * @param d2 The deck to discard to
     */
    public void atomicTurn(Deck d1, Deck d2) {
        // utilises drawCard() and dicardCard() methods
    }

    /**
     * Checks the players cards to see if they have a winning hand.
     * Does not have to be preferred denomination, as long as there is 4 cards of the same value
     * @return True if the player has a winning hand
     */
    public Boolean winCondition() { return false; }

    /**
     * @return The player's hand as an array of Cards
     */
    public Card[] getHand() { return new Card[0]; }
}
```

Figure 5: Player class outline

The Player class *(Fig. 5, left)* defines drawCard and discardCard methods, which will be contained within the atomicTurn method. This effectively makes the drawing/discarding of a card an atomic action, so outside of this, the Player's hand will always have a constant number of cards. It also contains a method to check whether the player's current hand is a winning hand.

Each instance also contains a FileWriter instance – this is to allow player actions to be documented in an output file.

```java
import java.io.FileWriter;

public class PlayerThread extends Thread {
    private Thread MAIN_THREAD;
    private FileWriter fw;
    private Player player;
    private Deck drawDeck;
    private Deck discardDeck;
    public PlayerThread(Thread MAIN_THREAD, FileWriter fileWriter, Integer pId,/
        Card[] playerHand, Deck draw, Deck discard) {}

    /**
     * Thread mainloop - runs player's atomic turn, checks for win condition;
     * If player has won, set CardGame.winPlayer to PlayerID and interrupt player thread;
     * Also writes win anouncements / final hands to the player's output file
     */
    public void run() {}

    /**
     * @return Player associated with thread
     */
    public Player getPlayer() { return player; }
}
```

Figure 6: PlayerThread class outline

The PlayerThread class *(Fig. 6, left)* will encapsulate an instance of the Player class, and implement the threaded behaviour, calling atomic turn method and checking the win condition, notifying the main thread when this is met.

Testing

For our tests, we will be using JUnit 4 (specifically 4.13.2).

Each of the supporting classes has a range of unit tests written for them to test their methods and behaviours. CardGameTestSuite class acts as a wrapper for the rest of the tests, running all classes.

For Card, this is simply getters and setters – the constructors are tested within the getter test. Certain duplicates have also been introduced to double-check for any unexpected values.

Similarly for Deck, however this class also encapsulates drawing/discarding functionality which must be tested. As cards are always drawn from the top and discarded to the bottom, it is easy to check the value of card outputted, and the state of the residual deck – this is all deterministic behaviour.

For Player, we set up some mock decks/hands with known card values, as this creates somewhat predictable behaviour (except for card discarding, which has a level of randomness – this is addressed in separate tests which uses the card's "age" attribute) and allows us to test the atomicTurn() method by asserting the number of cards in each deck/the players hand after the turn, and any cards of preferred denomination which should be retained *(see Fig. 7, below)*.

```
@Test
public void testAtomicTurn_discardByAge() throws IOException, InterruptedException {
    Deck d1 = new Deck(new Card[]{new Card(3)});
    Card[] cards = new Card[4];
    for(Integer i=0;i<4;i++) { cards[i] = new Card(2); }
    FileWriter f = new FileWriter("test_out.txt");
    Player p = new Player(f, 1, cards);
    for(Integer i=0; i<8; i++) {
        for(Card c : p.getHand()) { c.resetAge(); }
        d1.getDeck()[0].age();
        d1.getDeck()[0].age();
        p.atomicTurn(d1, d1);
        assertEquals("discardCard() failed: oldest card not discarded", 3, (int)d1.getDeck()[0].getValue());
    }
}
```

Figure 7: testAtomicTurn_discardByAge()

```
@Test
public void testAtomicTurn() throws IOException, InterruptedException {
    Card[] cards = new Card[]{new Card(1), new Card(2)};
    Deck d1 = new Deck(cards);
    Deck d2 = new Deck(cards);
    Card[] hand = new Card[4];
    for(Integer i=0;i<4;i++) { hand[i] = new Card(i+1); }
    FileWriter f = new FileWriter("test_out.txt");
    Player p = new Player(f, 1, hand);
    p.atomicTurn(d1, d2);
    assertEquals("Draw unsuccessful", d1.getDeck().length, 1);
    assertEquals("Discard unsuccessful", d2.getDeck().length, 3);
    assertEquals("Hand card count mismatch", 4, p.getHand().length);
    assertEquals("Discard condition failed: card with value 1 should be retained",
        1, (int)p.getHand()[3].getValue());
    assertNotEquals("Discard condition failed: card with value 1 should be retained",
        1, (int)d2.getDeck()[2].getValue());
}
```

Figure 8: testAtomicTurn()

In a variation on this test, the ages of cards within a deck/players hand are manipulated such that a known card is discarded each time *(see Fig. 8, left)*. Also, some cases contain assertions to check if preferred denominations of card (matching the player's id) are retained.

Testing the win condition was also key, so various concrete winning and losing scenarios are checked in a unit test *(testWinCondition() is not documented here for brevity)*, as well as induced winning states *(Fig. 9, right)*.

```
@Test
public void testAtomicTurn_Win() throws IOException, InterruptedException {
    Card[] deck = new Card[]{new Card(1), new Card(2)};
    Deck d1 = new Deck(deck);
    Card[] cards = new Card[4];
    cards[0] = new Card(2);
    for(Integer i=1;i<4;i++) { cards[i] = new Card(1); }
    FileWriter f = new FileWriter("test_out.txt");
    Player p = new Player(f, 1, cards);
    p.atomicTurn(d1, d1);
    assertTrue("Player turn failed to induce a winning state", p.winCondition());
}
```

Figure 9: testAtomicTurn_Win()

Reflection is used to individually test both the drawCard() and discardCard() methods by assertions on the residual hands given a known set of input cards *(Fig. 10 and Fig. 11, below)*.

```
@Test
public void testDrawCard() throws IOException, NoSuchMethodException {
    Deck d1 = new Deck(new Card[]{ new Card(1) });
    Card[] cards = new Card[4];
    for(Integer i=0;i<4;i++) { cards[i] = new Card(2); }
    FileWriter f = new FileWriter("test_out.txt");
    Player p = new Player(f, 1, cards);

    Method _drawCard = Player.class.getDeclaredMethod("drawCard", Deck.class);
    _drawCard.setAccessible(true);
    try {
        _drawCard.invoke(p, d1);
    } catch (Exception e) {}
    _drawCard.setAccessible(false);

    Card[] resultingHand = p.getHand();
    assertEquals("drawCard() failed: hand does not contain 5 cards", 5, (int)resultingHand.length);
    for(Integer i=0; i<4;i++) {
        assertEquals("drawCard() failed: residual deck mismatch", 2, (int)resultingHand[i].getValue());
    }
    assertEquals("drawCard() failed: residual deck mismatch", 1, (int)resultingHand[4].getValue());
}
```

Figure 10: testDrawCard()

```
@Test
public void testDiscardCard() throws IOException, NoSuchMethodException {
    Card[] cards = new Card[5];
    for(Integer i=0;i<4;i++) { cards[i] = new Card(2); }
    cards[4] = new Card(1);
    FileWriter f = new FileWriter("test_out.txt");
    Player[] players = new Player[]{ new Player(f, 1, cards), new Player(f, 2, cards) };

    Method _discardCard = Player.class.getDeclaredMethod("discardCard");
    _discardCard.setAccessible(true);
    try {
        for( Player p : players ) {
            _discardCard.invoke(p);
        }
    } catch (Exception e) {}
    _discardCard.setAccessible(false);

    Card[] resultingHand;
    for(Integer i : new Integer[]{ 1,2 }) {
        resultingHand = players[i-1].getHand();
        assertEquals("discardCard() failed: hand does not contain 4 cards", 4, (int)resultingHand.length);
        for(Integer j=0; j<3; j++) {
            assertEquals("discardCard() failed: residual deck mismatch", 2, (int)resultingHand[j].getValue());
        }
        assertEquals("discardCard() failed: residual deck mismatch", (int)i, (int)resultingHand[3].getValue());
    }
}
```

Figure 11: testDiscardCard()

The PlayerThread tests lead on from these. Again, this class contains a getter method for accessing the player instance which must have an associated unit test. Outside of that, the run method must ensure that the contained player's hand must contain 4 cards throughout, so the thread is run for 500ms, and the size of the player's hand is checked following that *(see Fig. 12, below)*.

```
@Test
public void testRun() throws IOException {
    Card[] cards = new Card[10];
    for(Integer i=0;i<10;i++) { cards[i] = new Card(i+1); }
    Deck d1 = new Deck(cards);
    Card[] hand = new Card[4];
    for(Integer i=0;i<4;i++) { hand[i] = new Card(i+1); }
    FileWriter f = new FileWriter("test_out.txt");
    PlayerThread pt = new PlayerThread(Thread.currentThread(), f, 1, hand, d1, d1);
    pt.start();
    try {
        Thread.sleep(500);
    } catch (Exception e) {}
    pt.interrupt();
    assertEquals("run() failed: Player's hand does not contain 4 cards", 4, pt.getPlayer().getHand().length);
}
```

Figure 12: testRun()

```
@Test
public void testRun_Win() throws IOException {
    Card[] cards = new Card[4];
    Card[] hand = new Card[4];
    for(Integer i=0;i<4;i++) {
        cards[i] = new Card(1);
        hand[i] = new Card(2+i);
    }
    Deck d1 = new Deck(cards);
    FileWriter f = new FileWriter("test_out.txt");
    PlayerThread pt = new PlayerThread(Thread.currentThread(), f, 1, hand, d1, d1);
    pt.start();
    try {
        Thread.sleep(500);
    } catch (Exception e) {}
    assertTrue("Player failed to win game", pt.getPlayer().winCondition());
}
```

Figure 13: testRun_Win()

The Player "strategy" is externally tested by a unit test here which runs the thread for a further 500ms (or until interrupted) using a deck containing only a winning hand – the resulting state of the program is such that a winning condition for the player is effectively guaranteed, and this is checked using assertions as usual *(see Fig. 13, left)*.

We believe that the code coverage for our supporting classes is effectively 100%.

As CardGame only holds the main code, there are no methods to test per se. However, we can run the main loop, and check the residual static variables, as this will verify a significant proportion of the code. By doing this, we also have a chance to check the input verification: the player count input is tested on letters, negative, zero and positive values; the pack filepath input is tested on non-existent, as well as valid/invalid files *(see Fig. 14, right)*.

```
@Test
public void testCardGame() throws IOException {
    String inputs = "-1\n0\n4\nnotapack.txt\nbin/pack.txt";
    InputStream testInputStream = new ByteArrayInputStream(
        inputs.getBytes(StandardCharsets.UTF_8));
    System.setIn(testInputStream);
    CardGame.main(null);
    assertNotEquals(0, (int)CardGame.winPlayer);
    assertEquals(4, (int)CardGame.players);
    assertEquals("bin/pack.txt", CardGame.packFile);
    testInputStream.close();
    CardGame.winPlayer = 0;
    CardGame.players = 0;
    CardGame.packFile = "";
}
```

Figure 14: testCardGame(); duplicate omitted