CS342: Software Design

September 26, 2017
Agenda

Project 1 discussion:

- Overview
- Code examples
- Class designs

Testing for Project 1

Different levels of testing
Card Game: Poker - 5 Card Draw

1. A Card Class - This class will keep the information about each card.
2. A Card Pile Class - This class is a collection of cards. While this class cannot just be an instance of a Java Collection Class, you may use a Java Collection Class instance as a data member of the Card Pile Class.
3. A User Player Class - This class will keep track of the user's hand and interact with the user interface.
4. An Opponent Player Class - This class will keep track of the computer player's hand and interact with the Artificial Intelligence.
5. A Game Class - This class is to contain the method main. This class will also keep track of the particulars of the game, such as the order of actions of the game (shuffling, dealing, discarding and determining the winner) and the evaluation of the hand (does the player have a Straight Flush, Four of a Kind, Full House, etc.). Actually it might be better to divide this into two classes: one to deal with the order of the actions and one to evaluate the hands.
You may get this when building large scale software...
Cleaner class design

**Player Base class:**
common fields and methods for user and computer
- Add a card to hand
- Discard a card
- Has flus, has four suit, has etc
- Calculate rank
- Cal # of cards u can discard
- Get hand

**Five CardDraw (Main):**
UI and orchestration
- Prompt # of players
- Trigger dealing
- User's turn
- AI's turn
- Display results

**Game Session: main logic and workflow**
- Set up pile
- Instantiate Players
- Deal to Players
- Players replace cards
- Decide results and winners

**Computer Player**
- AI function

**User Player**
- Interactive selection

**Card Pile**
- Shuffle
- Deal a card

**Card**
- Number, suit, string
Main function class: orchestration

```java
public class FiveCardDraw {

    public static void main(String[] args) {
        GameSession gameSession = new GameSession();
        int numPlayers = promptPlayerNumber();
        gameSession.setPlayerNumber(numPlayers);
        System.out.println("Starting Game. Dealing cards to " + numPlayers + " players");
        gameSession.dealToAllPlayers();
        displayHand(gameSession.getUserPlayerHand());
        displayReplaceInstruction(gameSession.getUserReplaceChances());
        gameSession.userPlayerReplaceCards();
        System.out.println("Your turn ends. Computer's turn");
        gameSession.computerPlayersReplaceCards();
        gameSession.calculateRanking();
        displayResults(gameSession.getPlayersSortedByRanking());
    }
}
```
Main function class: UI

```java
private int promptPlayerNumber() {
    System.out.println("Welcome to Five Card Draw Game. Please enter number of players: ");
    Scanner scanner = new Scanner(System.in);
    int input2;
    if (scanner.hasNextInt()) {
        while (((numPlayers = scanner.nextInt()) > 3 || (numPlayers < 1))) {
            System.out.println("Error. Please input a number between 1-3");
        }
    } else {
        System.out.println("Error. Input is not a number. Exiting");
    }
    return input2;
}

private void displayHand (Card[] cards) {
    System.out.println("Your hand currently:");
    for (Card card : cards) {
        //display card and index
    }
}

//this should be refactored. UI is containing logic
private void displayReplaceInstruction(int numChances) {
    if (numChances == 4) {
        System.out.println("Since you have an Ace, you can keep the Ace and discard the other");
    } else {
        System.out.println("You can discard up to 3 cards");
    }
    System.out.println("List the card numbers you wish to discard. Type 'x' when done");
}

private void displayResults (Result[] results) {
    for (Result result : results) {
        //print out result by rank, including type, hand, etc
    }
}
GameSession class: game level logic and workflow

```java
public class GameSession {
    private int numPlayers;
    private UserPlayer userPlayer;
    private ComputerPlayer [] computerPlayers;
    private SessionStatus status; // Enumeration

    public GameSession() { // do some initialization here
    }

    public void SetPlayerNumber(int num) {
    }

    public void DealToAllPlayers() {
    }

    public void UserPlayerReplaceCards() {
    }

    public void ComputerPlayersReplaceCards() {
    }

    public void CalculateRanking() {
    }

    public Result [] GetPlayersSortedByRanking() {
    }
}
```
Player classes

```java
public class BasePlayer {
  private int curNum = 0; // current number of cards in hand
  private String name;
  private Card[] hand = new Card[5]; // array of Card to hold all cards in hand

  public BasePlayer(String _name) {
    name = _name;
  }

  public void ResolveReplaceChances() {
    // logic...
  }

  public void ReplaceOneCard(int index, Card newCard) {
    this.hand[index] = newCard;
  }

  public int ScanAndValidateCardToDiscard() {
    // logic...
  }

  public void addCard(Card c) {
    // logic...
  }

  public boolean hasAce() {
    // logic...
  }

  public boolean hasFlush() {
    // logic...
  }

  public boolean hasFourSuit(CardPile pile, ArrayList<Card> discardPile) {
    // logic...
  }

  public boolean hasFourStraight(CardPile pile, ArrayList<Card> discardPile) {
    // logic...
  }

  public boolean hasStraight() {
    // logic...
  }

  // Hand is a Four-of-a-Kind. Discard the odd one out
  public boolean hasFour(CardPile pile, ArrayList<Card> discardPile) {
    // logic...
  }
}
```

```java
public class UserPlayer extends BasePlayer {
  public int ScanAndValidateCardToDiscard() {
    boolean validInput = false;
    while (!validInput) {
      // scan for input
      // if input is "X", break and return -1
      // if input is between 1 and 5, break and return input
      // otherwise repeat
    }
  }
}
```

```java
public class OpponentPlayer extends BasePlayer {
  public int AISelectCardToDiscard() {
    // 1. First check if the computer player has more cards
    // 2. If the hand evaluates to "HighCard", determine
    // 3. Next determine if the user has 4 cards
    // 4. Next if the user has an Ace, discard the other
    // 5. Otherwise, keep the two highest cards
  }
}
```
Card Class

```java
public class Card {

    private int number;  // number of the card
    private Suit suit;   // suit of the card
    private static String[] Num = {"A","2","3","4","5","6","7","8","9","10","J","Q","K"};

    public Card(Suit s, int _number) {
        this.suit = s;
        this.number = _number;
    }

    public String getString() {
        return Num[number] + suit.toString();
    }

    public String getSuit() {
        return this.suit.toString();
    }

    public int getNum() {
        return this.number;
    }
}
```

Good job with the enum!

What test cases do we need here?
Let’s take another look: Class dependency

Diagram:
- Game
  - ComputerPlayer
  - UserPlayer
  - CardPile
- Card

Dependency relationships:
- Game depends on ComputerPlayer, UserPlayer, and CardPile.
- ComputerPlayer depends on Game and Card.
- UserPlayer depends on Game and Card.
- CardPile depends on Game and Card.
And Cleaner class design

**Player Base class:**
- common fields and methods for user and computer
  - Add a card to hand
  - Discard a card
  - Has flus, has four suit, has etc
  - Calculate rank
  - Calculate # of cards u can discard
  - Get hand

**Five CardDraw (Main):**
- UI and orchestration
  - Prompt # of players
  - Trigger dealing
  - User's turn
  - AI's turn
  - Display results

**Game Session: main logic and workflow**
- Set up pile
- Instantiate Players
- Deal to Players
- Players replace cards
- Decide results and winners

**Computer Player**
- AI function

**User Player**
- Interactive selection

**Card Pile**
- Shuffle
- Deal a card

**Player Base class:**
- Add a card to hand
- Discard a card
- Has flus, has four suit, has etc
- Calculate rank
- Calculate # of cards u can discard
- Get hand

**Card**
- Number, suit, string
Create Loosely coupled, self-contained classes

- Less dependent on other class
- Can perform its own domain of business
- Respect other’s boundary. Don’t overreach
- Separate UI vs. logic, and logic vs. data access
- One direction dependency.
- Layer: lower layer shouldn’t be aware of higher layer. High layer shouldn’t care how lower layer does its job
- Why separate frontend and backend classes?
- Easier to debug, read, maintain, and scale.

What could have been done differently?

- Player show hands: move display part out of Player class
- “Hand” class?
What are we missing here?

```java
public class CardPile {
    // data structure used to hold collection of cards
    private ArrayList<Card> pile;
    // default constructor
    public CardPile(){
        pile = new ArrayList<Card>();
        for(int a = 0; a < 3; a++){
            for(int b = 0; b < 12; b++){
                pile.add(new Card(Suit.values()[a], b));
            }
        }
        // shuffle the deck before dealing to players
        System.out.println("The cards are being shuffled");
        Collections.shuffle(pile);
    }
    public Card draw(){
        return pile.remove(pile.size() - 1);
    }
    public int gettotal(){
        return pile.size();
    }
}
```

```java
public class TestCardPile {
    CardPile cardPile = new CardPile();

    @Test
    public void testCreateNewPile() {
        assertEquals(52, cardPile.gettotal(), 0.0);
        // Is this enough?
    }
    @Test
    public void testDraw() {
        int beforeNumber = cardPile.gettotal();
        Card card = cardPile.draw();
        int afterNumber = cardPile.gettotal();
        assertEquals(beforeNumber, afterNumber + 1, 0.0);
    }
```
CardPile test continues

How do we test the drawn card should match the missing card from pile?

How do we access private methods and fields of CardPile?
BasePlayer Test

Test “hasAce”:
- Both positive and negative cases
- How many times do we need to run
- Same for hasFourSuit,

ResolveReplaceChances: need to test path hasAce and not haveAce

Hint: use @Before, @After to set up hands

\[\text{addCard()} \rightarrow \text{hasAce()} \rightarrow \text{ResolveReplaceChances()}]\]
Different levels of testing

- UAT: real users
- System testing: QA
- Integration: Developers, maybe QA
- Unite testing: developers
More about testing

- System testing vs. functional testing
- Regression testing
- Smoke testing
- Continuous testing
- Load/performance testing
- Edge case (what happen to GameSession if two players have same hand rank? What if someone’s birthday is right at 0:00AM? What if...?)