Object-Oriented Programming

Technical Implementation
by Ronin Brookes

Final-Year student, 2023

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Table of Contents

Problem Statement

Design an application with the following functionality:

• Simulate shuffling a standard deck of 52 cards.

• Deal a single hand of 5 cards to the player.

• Evaluate the player's hand, informing them of the highest ranked poker hand that matches their hand of 5 cards.

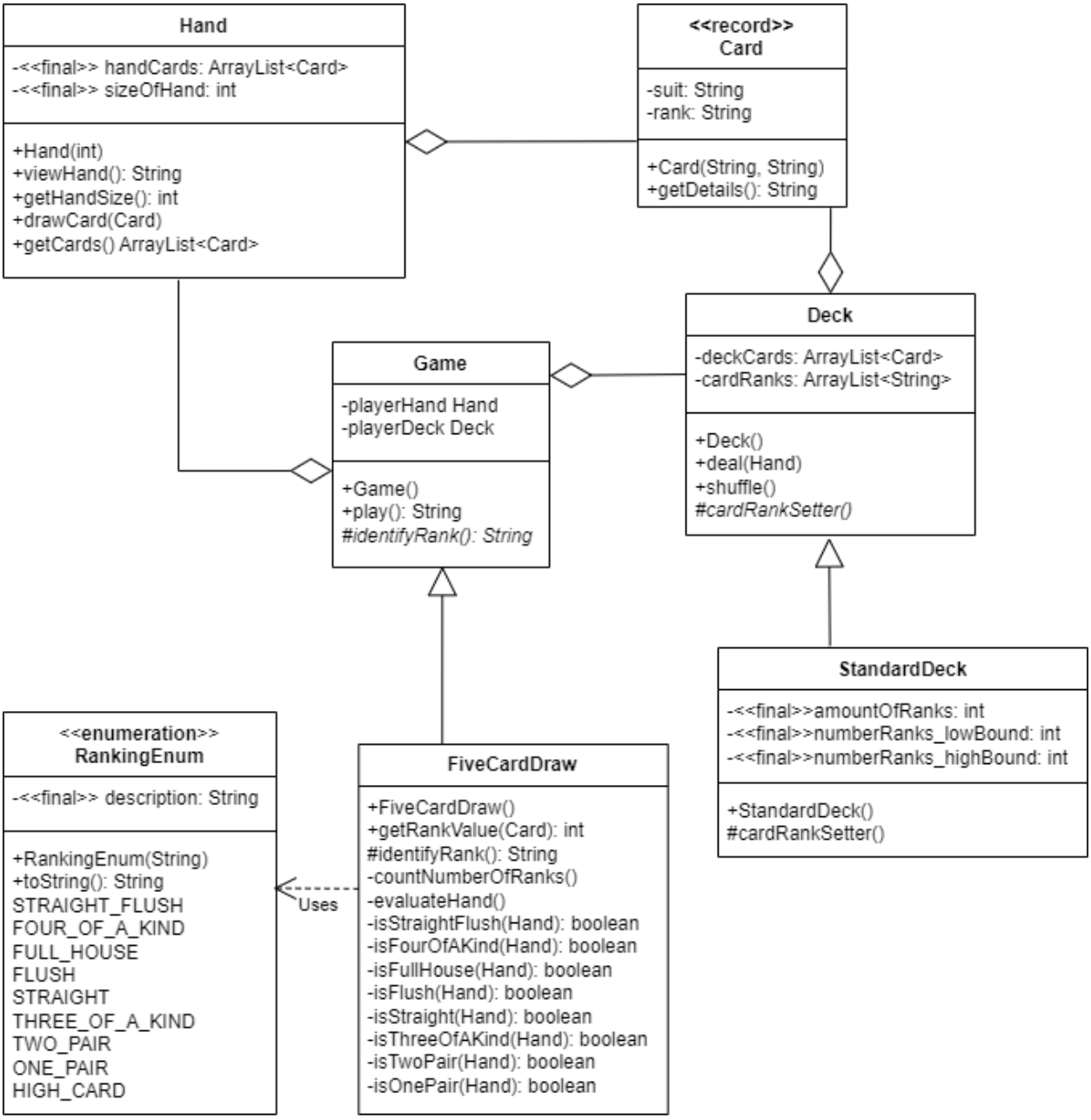
The ranks are as follows:

1. Straight Flush
2. Four of a Kind
3. Full House
4. Flush
5. Straight
6. Three of a Kind
7. Two Pair
8. One Pair
9. High Cards

Chosen Solution

For this task I have implemented a Spring Boot Application in Java. Spring Boot is a Java framework that simplifies the process of building production-ready, stand-alone, and highly maintainable Spring based applications. It also has embedded web servers, removing the need to set up and configure a separate web server for web applications. This application will be console-based but is designed with the assumption that it can in future be extended to be a web application. Spring Boot aids in simplifying this process.

UML Diagram


Hand
-<<final>> handCards: ArrayList<Card>
-<<final>> sizeOfHand: int
+Hand(int)
+viewHand(): String
+getHandSize(): int
+drawCard(Card)
+getCards() ArrayList<Card>
<<record>>
Card
-suit: String
-rank: String
+Card(String, String)
+getDetails(): String
Deck
-deckCards: ArrayList<Card>
-cardRanks: ArrayList<String>
+Deck()
+deal(Hand)
+shuffle()
#cardRankSetter()
Game
-playerHand Hand
-playerDeck Deck
+Game()
+play(): String
#identifyRank(): String
StandardDeck
-<<final>>amountOfRanks: int
-<<final>>numberRanks_lowBound: int
-<<final>>numberRanks_highBound: int
+StandardDeck()
#cardRankSetter()
<<enumeration>>
RankingEnum
-<<final>> description: String
+RankingEnum(String)
+toString(): String
STRAIGHT_FLUSH
FOUR_OF_A_KIND
FULL_HOUSE
FLUSH
STRAIGHT
THREE_OF_A_KIND
TWO_PAIR
ONE_PAIR
HIGH_CARD
Uses
FiveCardDraw
+FiveCardDraw()
+getRankValue(Card): int
#identifyRank(): String
-countNumberOfRanks()
-evaluateHand()
-isStraightFlush(Hand): boolean
-isFourOfAKind(Hand): boolean
-isFullHouse(Hand): boolean
-isFlush(Hand): boolean
-isStraight(Hand): boolean
-isThreeOfAKind(Hand): boolean
-isTwoPair(Hand): boolean
-isOnePair(Hand): boolean

Design Explanation

Maintainability:

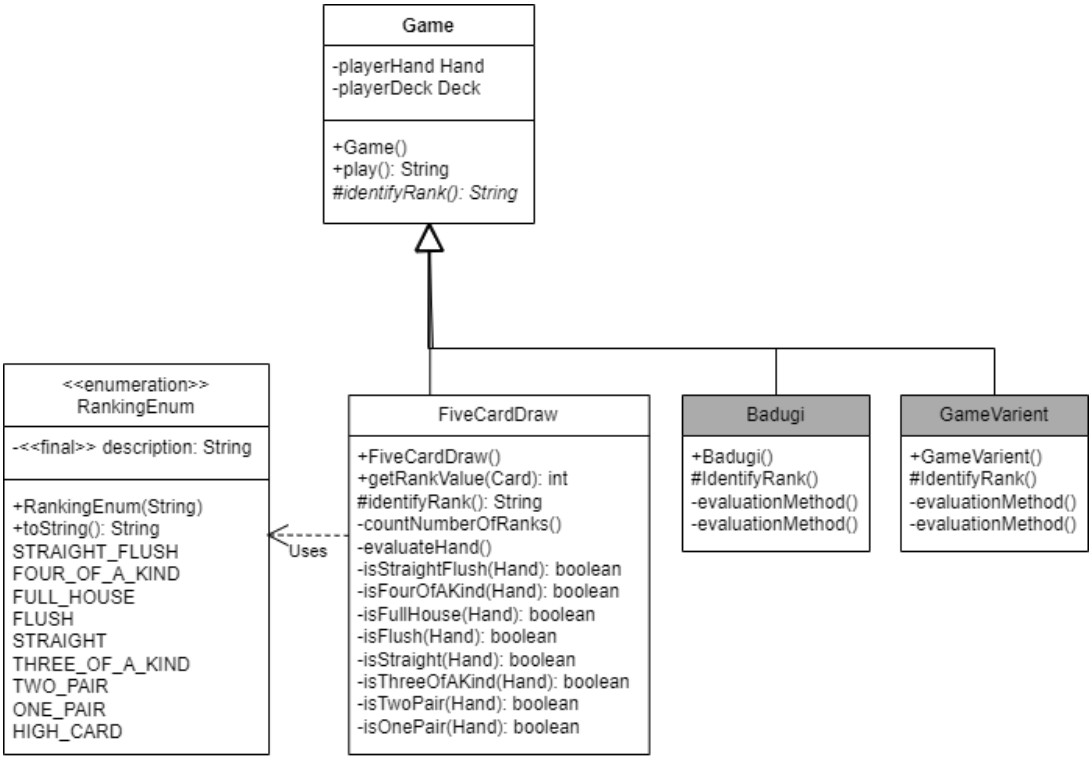
The application is made maintainable with modularization, in which the program is separated into 3 base classes,2 derived classes, 1 report class and 1 enumeration class. Each of these serve a specific purpose.

- The Card record class is used to store and retrieve card ranks and suits.
- The Hand class manages the card objects the player holds.
- The Deck class manages the card objects the deck holds, including shuffling and dealing cards from the deck to a player's hand.
- The StructuredDeck class is derived from the Deck class. It is responsible for constructing the specific deck tailored to its name.
- The Game class initiates the specific game and calls the relevant primitive function to have the game play out a specific way. (Shuffle-Deal-IdentifyRank).
- The FiveCardDraw class is one of these specific games. It is a variant of the many types of games this application can be extended to support. It is used with the enumeration class for evaluating the player's hand, identifying, and returning a result based on the rules of the specific game of the same name.

This results in a separation of concerns - smaller modules that are easier to manage, read and understand, as well as increasing ease of maintenance, with potential bugs being isolated and thus easier to find and fix. Smaller modules are also easier to unit test. Modules encapsulate their internal details, limiting exposure to specific public interfaces, keeping the data safe from side effects, and enhancing security. Bounds checking and unit testing is also implemented, resulting in more robust, maintainable code that helps prevent future developers from writing faulty variant subclasses.

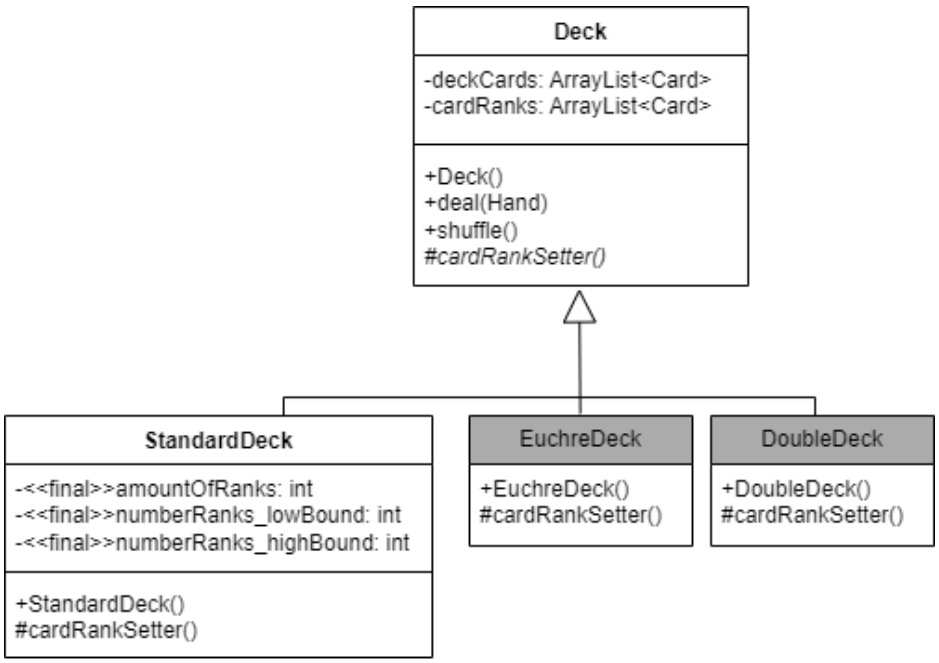
Extensibility:

The application is made extensible by use of modularity as explained above as well as the open-closed principle, in which classes are made to be open to extension but closed for modification. This allows you to easily add new features to the application by adding new classes or modules rather than altering existing ones. To illustrate this, one can investigate structure of the Game class:



Whenever a developer wants to add another game variant, all that is needed is to add a class per variant. Each of the subclasses has its own constructor and implementation of the primitive identifyRank() function, which will evaluate a player's hand in a unique way and return different results depending on the hardcoded games rules. If needed, a developer can implement a corresponding enumeration class as well.

Another way this principle is supported in the application is with the Deck class:

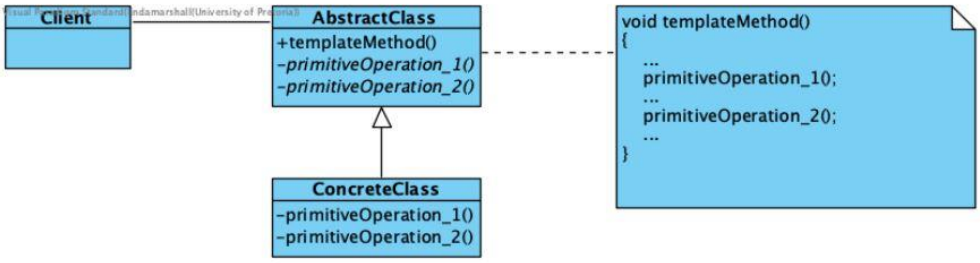


Whenever a developer wants to add a new variant of the deck, such as a DoubleDeck, or EuchreDeck, they can do so by simply adding another class with its own constructor, and implemented cardRankSetter() function, which will instantiate all cards of the deck to the specific deck formula and rank values.

It should be noted that the same concept can be applied to cards, in that a developer could add variants of cards with specific traits. For the sake of simplification, this application in its current state only operates with cards that have a String suit and a String rank. The string values of the suit and rank are however fully up to the developer and relates entirely to how they wish to instantiate the cards within the specific deck variant class. It should also be noted the value of the ranks is also only implemented in the game variant classes. Also note that all classes shown in the UML diagrams above with grey names are hypothetical and not actually implemented in the application.

Use of Design Patterns

Template Method:



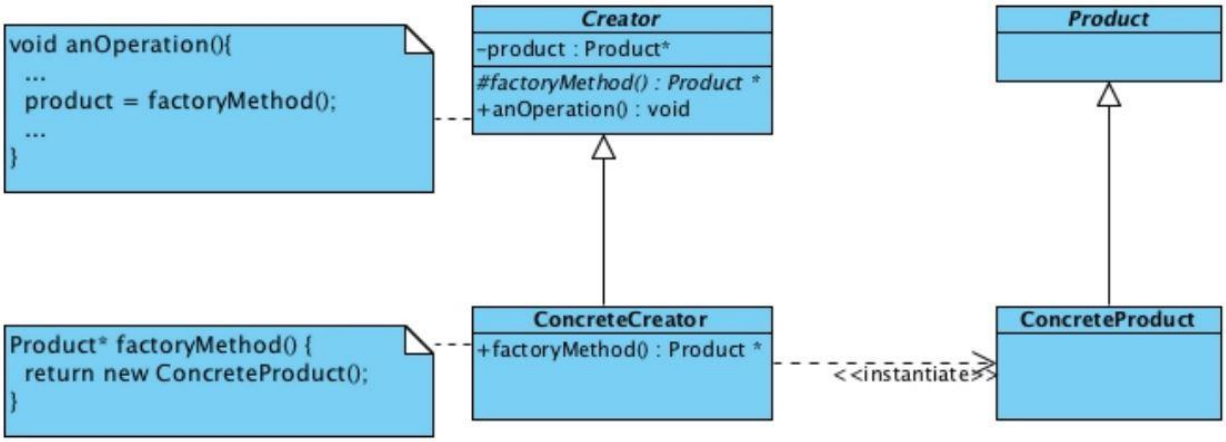
The Game class takes the role of the AbstractClass participant, with the variants of the game, namely FiveCardDraw, being the ConcreteClass participant. The play() function is our templateMethod(), which will call the primitive function, identifyRank(), which is abstract and thus only implemented in the subclasses (variants) of the Game class. This function will evaluate the players hand uniquely to its own hardcoded rules once called.

The Template Method improves the application in that in promotes reusing code, as well as eases testing, increases flexibility and forces consistency.

```
//Template method to call the specific Identify() function:
3 usages  new *
public final String play() {
        playerDeck.shuffle();
        playerDeck.deal(playerHand);
        System.out.println(playerHand.viewHand());
        return identifyRank();
}
```

```
//Evaluate the hand for each possible ranking:
2 usages  new *
@Override
protected String identifyRank() {
    if(playerHand.getCards().size()!=5){
        String errorMessage = "identifyRank(): Hand size for FiveCardDraw is not 5 cards";
        logger.log(Level.SEVERE, errorMessage);
        throw new IllegalArgumentException(errorMessage);
    }
    countNumberOfRanks();
    return evaluateHand().toString();
}
```

A play on the Factory Method



The application does not directly follow the Factory Method model, yet still maintains some of its benefits without using a traditional anOperation() or factoryMethod() stand in. Instead, it uses polymorphism to evoke the StandardDeck subclass constructor which instantiates a list of cards to a hardcoded formula, and this formula is subclass specific, meaning any number of deck type classes can instantiate a list of cards to any custom formula, much like a factory method implementation. Benefits of this approach include separation of concerns, flexibility, and encapsulation of object creation logic.

```
//Initialize deck cards:
3 usages  new *
public StandardDeck() {
    deckCards = new ArrayList<>();
    for (int i = 0; i < amountOfRanks; i++) {
        deckCards.add(new Card(cardRanks.get(i), suit: "Hearts"));
        deckCards.add(new Card(cardRanks.get(i), suit: "Diamonds"));
        deckCards.add(new Card(cardRanks.get(i), suit: "Clubs"));
        deckCards.add(new Card(cardRanks.get(i), suit: "Spades"));
    }
}
```

Use of Data Structures

ArrayList

An ArrayList is used in the Hand class and the Deck class, to hold hand cards and deck cards respectively. The reason ArrayLists were used is because the number of cards needed for a hand and deck is not known at compile time and depends on the specific game played. The game chosen, in this case, FiveCardDraw, instantiates its Hand object with a specific size, and then instantiates its Deck object with that Hand object. The Deck object, with the given Hand object, initializes its deckCards ArrayList with a specific size. In the deal(Hand playerHand) function, cards are added to the handCards ArrayList and removed from the deckCards ArrayList. The benefits of an ArrayList for this application are:

- Dynamic sizing for unknown hand and deck card array sizes.
- Efficient random access for quickly accessing elements especially useful in the identifyRank() implementation where individual hand cards are assessed.
- Easy element manipulation, useful when needing to shuffle the deck cards.

HashMap

In the FiveCardDraw class, a HashMap is used to count the number of each different rank in the cards in the player's hand. So, for specific helper functions determining a rank, they can use this HashMap if they need to evaluate this number. The benefits of a HashMap for this application are:

- Dynamic sizing since the amount of hand cards are not known at compile time.
- Efficient search, useful since these helper functions often make one search call.

- **Record**

One of the major assumptions made in the design process was that cards to be evaluated will be done so on rank and suit alone, thus no need for inheritance was needed for specific type of cards, rather the developer can create their own custom cards with their own types of ranks and suits and rules to go along with them. For example, a joker card can simply have “Joker” as both its suit and rank and have the developer assign their own numerical value of this card in a specified subclass, either inherited from Deck or Game.

A record is ideal for this purpose as they are primarily used for simply storing and retrieving data. They are especially useful for this application as they are immutable, in that their state can never change, as well as easily readable, requiring little code to express a lot but not too much to the point it hurts readability.

- **Enumeration**

To abstract and encapsulate the various evaluation results, a specialized form of data structure known as an enumeration was used. Enums were ideal for this purpose as they are generally used to represent lists with fixed named values, each of with are used to represent a potential ranking for the cards. This was added late in the design process as another method, involving several if statements and string returns were used instead, which worked as well, albeit less readable. For this reason, the use of an enum class is entirely optional to the developer, and they are free to evaluate the hand of cards any way they wish.

Assumptions

In implementing this solution, I assumed the following:

- Only one type of card, namely a card with a String suit and String rank, is used for all games played, however the nature of these String values is up to the developer implementing the specific deck subclass, as well as the rank number values. The reason for this design decision was to keep the focus of the application, at least for now, on the evaluation of cards based upon their ranks and suits, regardless of how those values are specified.
- The extensibility of the app pertains to the ability of adding limitless types of evaluation games, each with their own hand size and hand evaluation techniques, as well as with limitless types of decks, each with their own card list sizes, and formulas for instantiating those cards.
- Straight Flush occurs when the hand is ranked both Flush and Straight.
- Four of a kind occurs when the hand has 4 cards each of the same rank.
- Full house occurs when there are 2 cards of the same rank and 3 cards of the same, different rank.
- Flush occurs when all cards have the same suit.
- Straight occurs when all cards are in either increasing or decreasing rank order.
- Three of a kind occurs when there are 3 cards of the same rank.
- Two pair occurs when there are 2 pairs of the same ranked cards.
- One pair occurs when there is 1 pair of cards of the same rank.
- High card occurs when none of the above rankings occur.
- Rank values of the standard deck in increasing order is 2-10, J, Q, K, A.

Operation of the Application

- When the application starts a FiveCardDraw object is made via polymorphism.

```
//Initialize game type:
new *
public static void main(String[] args) {
    SpringApplication.run(CardGameApplication.class, args);
    Game mygame = new FiveCardDraw();
    System.out.println("\n----------------------------------------\n");
    System.out.println(mygame.play());
    System.out.println("\n----------------------------------------\n");
    System.exit( status: 0);
}
```

- The FiveCardDraw instantiates the playerHand object, and playerDeck object.

```
//Initialize hand and deck:
4 usages  new *
public FiveCardDraw() {
    System.out.println("Five-Card Draw Has Been Chosen!");
    playerHand = new Hand( sizeOfHand: 5);
    playerDeck = new StandardDeck();
}
```

- Upon creation, the playerDeck object specifies the ranking of the deck, making the construction of cards into the deck easier.

```
//Calls the method to set up the rankings of the cards of the deck:
1 usage  new *
public Deck() {
    cardRankSetter();
}
```

```
//instantiate and initialize the ranking for the cards of the deck:
1 usage  new *
@Override
protected void cardRankSetter(){
    cardRanks = new ArrayList<>();
    for (int i = numberRanks_lowBound; i < numberRanks_highBound; i++) {
        cardRanks.add(Integer.toString(i));
    }
    cardRanks.add("J");
    cardRanks.add("Q");
    cardRanks.add("K");
    cardRanks.add("A");
}
```

```
//Initialize deck cards:
3 usages  new *
public StandardDeck() {
    deckCards = new ArrayList<>();
    for (int i = 0; i < amountOfRanks; i++) {
        deckCards.add(new Card(cardRanks.get(i), suit: "Hearts"));
        deckCards.add(new Card(cardRanks.get(i), suit: "Diamonds"));
        deckCards.add(new Card(cardRanks.get(i), suit: "Clubs"));
        deckCards.add(new Card(cardRanks.get(i), suit: "Spades"));
    }
}
```

- Once all instantiation is done, from the CardGameApplication class, the FiveCardDraw object calls the play() function.
- The play() function calls the playerDeck object's shuffle and deal functions, shuffling the deck and dealing the cards to the playerHand object respectively.
- The play() function then calls the identifyRank() primitive function of a subclass.

```
//Template method to call the specific Identify() function:
3 usages  new *
public final String play() {
        playerDeck.shuffle();
        playerDeck.deal(playerHand);
        System.out.println(playerHand.viewHand());
        return identifyRank();
}
```

- The identifyRank() function evaluates the playerHand object and determines it's rank and returns the rank as a string value. It does this with several evaluation functions as well as an enum class to specify the exact ranking of the hand. Further details of this, like everything else, can be seen in the actual code with descriptive comments.

```
//Evaluate the hand for each possible ranking:
2 usages  new *
@Override
protected String identifyRank() {
    if(playerHand.getCards().size()!=5){
        String errorMessage = "identifyRank(): Hand size for FiveCardDraw is not 5 cards";
        logger.log(Level.SEVERE, errorMessage);
        throw new IllegalArgumentException(errorMessage);
    }
    countNumberOfRanks();
    return evaluateHand().toString();
}
```

Considerations

The following 3 aspects of the application were considered during the design process. They have not been implemented, but the application has been designed to accommodate each of them.

Replacing the shuffling algorithm with a more realistic alternative:

To have the shuffling algorithm made easily replaceable was to only have it implemented in one class, namely, the Deck base class, from which all deck variant subclasses will inherit. The class takes no parameters, and simply shuffles the existing deck cards for a deck object, regardless of its variant. Its only called in the play() template method in the Game base class, thus if in future a replacement algorithm were to be implemented, the developer would only need to alter code in 1 class, and if the name or parameters changed, 2 classes at most in the case of a name change. (Not accounting for unit tests, yet a name change would imply relevant changes in the tests as well.)

Accommodate Badugi, a different game variant, with different ranks and hand size:

To have different game variants, each with potentially different ranks, decks and hand sizes, I made 2 different types of subclasses, subclasses for games (Game class) and decks (Deck class). This ensures that various games with different decks can be used, since game subclasses instantiate both their hand object with a specific size as well as their deck object with a specific deck subclass. Each game subclass implements its own identify primitive function, establishing different ranks and rules per subclass. Each deck subclass can create whatever type of card collection it wants so long as the cards have

both a rank and suit. This results in an application that can easily cater for any poker variant, or any card evaluation game involving a manner or ranks and suits, including Badugi, regardless of hand size, rank, or even deck content.

Allowing for the application to be changed from console-based to web-based:

This application is a spring boot application that is at the moment only console-based but can be further implemented to be extended to a web-based application. Spring boot is used to build java web applications, by use of various tools including embedded web servers, web controllers and request mapping. A default RestController, meant to handle HTTP requests, map URLs to methods, and return responses, is outlined in the application but not yet implemented.