**PROJECT BACKGROUND AND DESCRIPTION**

The goal for this project is to create a finished, playable game of war. Users will be able to play with a computer or with a partner and use keyboard controls or text-based commands to continue the play. The game of war is simple (and official rules outlined by Bicycle cards can be found here https://bicyclecards.com/how-to-play/war/). A deck of cards is split evenly into two piles face down, one for each player. Both players flip up the top card of their deck at the same time and see what the face value on the card is (ace being high). The player with the higher-value card takes both cards and adds them to the bottom of their pile. In the case where both cards have the same value, each player then places another card face down and a third card face up. Whichever player’s new face-up card has the higher value takes all the cards placed down (6 at this point). This repeats for as long as the flipped up cards are equal in value.

The starting code we have for the project consists of four classes written in Java. The first class is a Card object class that creates a card that can be used to play. At the moment, it only has an overridden toString method, but the class will be updated to contain Suit and Value enum types. The second class, GroupOfCards, is an aggregation of Cards placed into an ArrayList to allow for a hand and a deck to be set for the game. This class contains a parameterized constructor, a shuffle method that uses Java’s Collections class, and mutator methods for the size data field. The third class is the Player class. This class has a constructor to create a Player object using a “playerID” string, mutator methods to access the playerID, and an abstract method called play that’s empty at the moment but will be updated or can be overridden to include the code logic for playing the game. Finally, the Game class is a model of the game for which we will create a child that has more specific rules and instantiates the methods. This class is so far built from a final gameName string and an ArrayList of Player objects. The parameterized constructor only needs a string for the name of the game and will instantiate the players ArrayList. Because of the final type of gameName, there is only a getter for the name of the game and not a setter. However the players list has a getter and a setter method. There is an abstract method called play in this Class as well, which is interesting considering the play method in the Player class. This can be overridden in the child class of Game to include more detailed instructions on how to display the game for the users and game logic. Finally, the class also contains a declareWinner abstract method which will contain the code to display the winner on the screen once the game is over.

The majority of the above classes are abstract classes with abstract methods which will allow for subclasses/child classes to be created that extend their superclass and override the methods to create an object-oriented program that takes advantage of polymorphism in Java’s language.

**PROJECT SCOPE**

Ahmad Bustani will be taking charge of writing and planning the main logic for the code, such as methods, data fields etc. Angela Villadiego will be responsible for taking charge of the high-level concepts and organizing the flow of the of the code. She will also be responsible for error checking, as well as organizing and note-taking at group meetings. Joshua Paivais is responsible for the design and output of the game and delegating tasks. Asha Sheikh will handle the debugging and error-checking/handling of the code. She will help edit the output of the game with Joshua and will also be in charge of scheduling and time management for the project.

The interface will be simple and at the moment we aren’t planning to give the cards a visual element, although that may be changed at a future date. Key-ins need to be recorded and a key from the right side (like the up arrow) needs to be configured to be for player 1 and a key from the left side (like the w key) needs to be configured to be for player 2. There needs to be a separate play method for human vs AI and human vs human. Output needs to be placed in a dialog box of window with the ability to reset play and display information textually and visually. Variables and methods to evaluate and change the number of cards in each players hands is necessary, as well as methods for running a “war” when the same card is flipped up and the next card should be face down rather than face up. A method to compare and decide the winner among two flipped up cards is needed as well.

**HIGH-LEVEL REQUIREMENTS**

War does not have many high level requirements in order to function, the basic concept of the game is to just compare two cards against each other and see which card is of a higher value, and then give the player who placed the higher value card both cards; whichever player ends up with the entire deck wins the game.

The requirements include:

1. Being able to register your name with the game.
2. Being able to register a match win by selecting the player with the highest value card in play and giving them both cards.
3. Being able to register a game win by detecting when one player has captured the entire deck and rewarding them with a point or a win message.
4. Displaying the deck sizes in real time in order to show the players who is winning at any given time.
5. Choosing between facing a player or a computer.

**IMPLEMENTATION PLAN**

<https://github.com/Paivajo/group-project.git>

Each developer needs to check the code every time before and after he wants to work on the project. NetBeans will be used to create the game and GitHub will help the group to complete the project together. We might need to use the debugging tools to help us make sure the game is working with no errors. Text files, diagrams, and the code of the game will be separated in folders so the group members who are working on the project don't get confused.

**DESIGN CONSIDERATIONS**

Encapsulation is used in the base code such as in the Player class where the playerID is a private data field but can be accessed using the mutator methods. Another example of this is within the Game class where the list of players can be accessed and set using mutator methods, but the gameName which is final can only be accessed. Delegation happens in the GroupOfCards class under the shuffle method. The process is actually delegated to the Collections Java class, which itself contains a shuffle method. GroupOfCards.shuffle() just passes the cards ArrayList into Collections.shuffle() to be shuffled. This is the only example of delegation that can be found in the base code given to us. Finally, flexibility and maintainability is a big part of the way this code is organized. The Card and Player objects are created in separate classes with abstract methods that can be overridden should there need to be a different kind of player or different kind of card.The same goes for the GroupOfCards object (which is dependent on the Card object) and the Game method (dependent on the Player object). Both of them contain abstract class play which can be defined by a subclass but called globally for the superclass and its subclass (using Object.play() for example will return something different if the object is a Player or a Game). This allows for easy manipulation of code and for code to be reused in different scenarios.