BlackJackChain

Documentation of Solidity and Smart Contract implementation

Bellucci Emanuele - 3088387
Bricarelli Paolo - 3000751
Bruno Luca - 3002713
Nuti Silvia - 1831031
Roscioli Dimitri - 3001293

Finance with big data, 2019/2020

Summary

Introduction 2

BlackJackChain as a smart contract 2

Structure of the code 2

Code: how it is written step by step 3

Contract – Owned 3

Contract – BlackJack 3

The RNG problem 9

Conclusion 10

References 10

Introduction

BlackJackChain is a blockchain based implementation of the casino game Blackjack. The Ethereum smart contract has been written in Solidity 0.4.26 exploiting the platform remix.ethereum.org. With the following report, we want to:

- explain the code step-by-step;
- give a valid documentation to ease comprehension;
- facilitate future extensions and improvements to the game.

In this document we will also explain the reasons behind the assumptions we hold when constructing the smart contract as well as an exhaustive explanation of the RNG problem.

BlackJackChain as a smart contract

Structure of the code

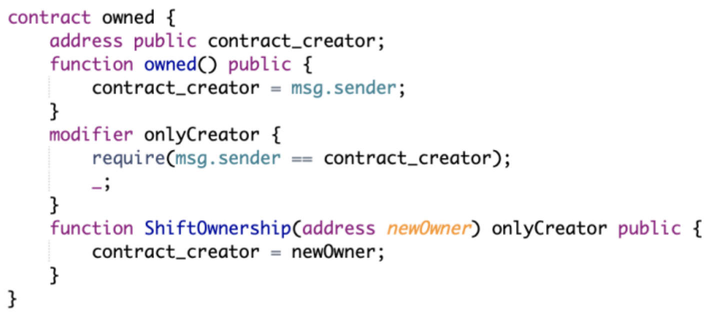
In this chapter we explain the code we implemented. You can find in the BlackJackChain.sol file uploaded in the following GitHub repository: https://github.com/Ziobero/Blockchain_based_blackjack.
Before diving deep into the code, we report here a list of contracts and functions defined in order to have a better understanding of the general structure of the code:

- **Contract Owned**
  - function owned
  - modifier onlyCreator
  - function ShiftOwnership

- **Contract Blackjack**
  - function GameBlackjack
  - function StartPlaying
  - function GenerateRandomNumber
  - function TransforNumberToCard
  - function giveCard
  - function Card
  - function Stand
  - function Value21forGamer
  - function BustedGamer
  - function payMe

Code: how it is written step by step

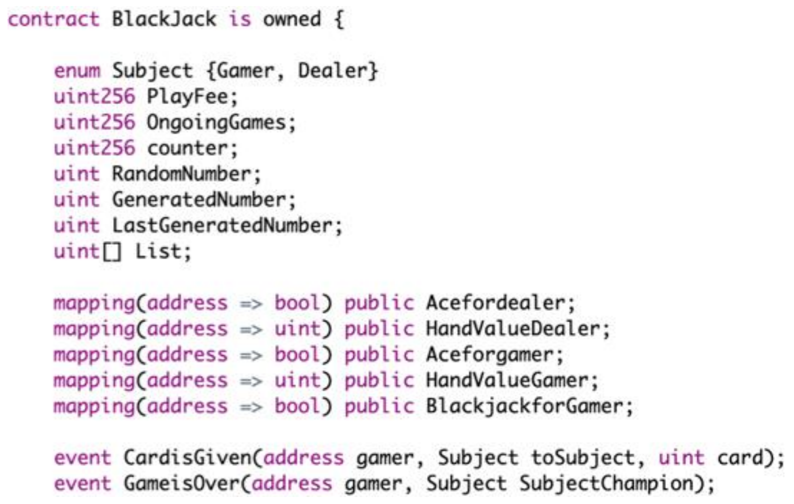
Contract – Owned



```
contract owned {
    address public contract_creator;
    function owned() public {
        contract_creator = msg.sender;
    }
    modifier onlyCreator {
        require(msg.sender == contract_creator);
        _;
    }
    function ShiftOwnership(address newOwner) onlyCreator public {
        contract_creator = newOwner;
    }
}
```

Initially we define a contract owned where we determine and set the owner of the contract as the message sender. We also require and ensure that only the owner is able to modify the contract.

Contract – BlackJack



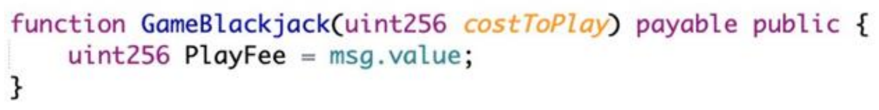
```
contract BlackJack is owned {

    enum Subject {Gamer, Dealer}
    uint256 PlayFee;
    uint256 OngoingGames;
    uint256 counter;
    uint RandomNumber;
    uint GeneratedNumber;
    uint LastGeneratedNumber;
    uint[] List;

    mapping(address => bool) public Acefordealer;
    mapping(address => uint) public HandValueDealer;
    mapping(address => bool) public Aceforgamer;
    mapping(address => uint) public HandValueGamer;
    mapping(address => bool) public BlackjackforGamer;

    event CardisGiven(address gamer, Subject toSubject, uint card);
    event GameisOver(address gamer, Subject SubjectChampion);
```

In the contract BlackJack we defined:

- the **entities** that will play the game;
- the **cost to play** a single hand;
- the **number of games in progress** (to guarantee that the casino has enough money in the contract to pay all potential winners);
- a **counter** (it will be used later on to generate a random number);
- a **random number** from 1 to 312;
- the **card given** to the subject;
- a **List** containing the previously generated numbers.

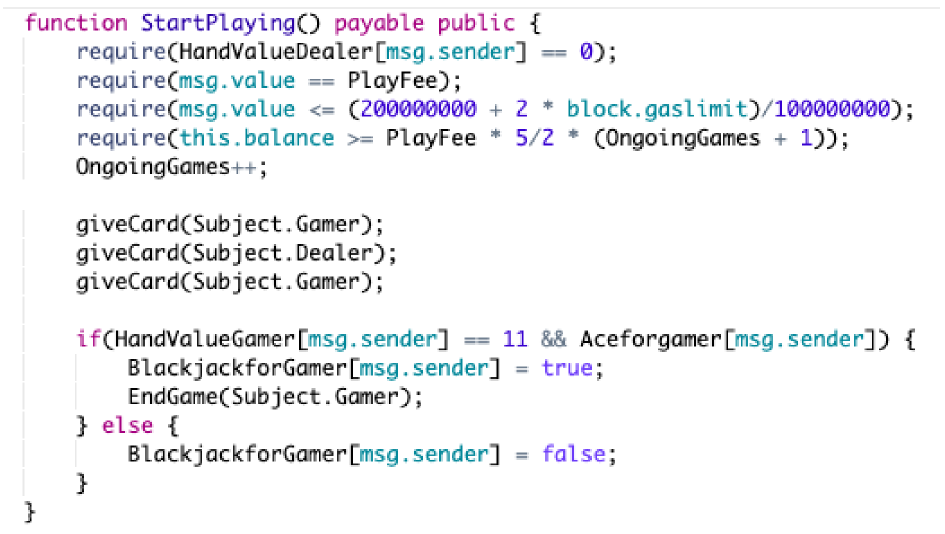
Mapping is like a dictionary that is used to keep track of certain steps and values of each game in progress. In ***Acefordealer*** and ***Aceforgamer*** we used bool because we just store whether the gamer or the dealer has an ace. Mapping is also used to store the score of the hand of the dealer and the gamer (***HandValueDealer*** and ***HandValueGamer***) as an unsigned integer and to keep track of whether the dealer has blackjack (***BlackjackforGamer***) as a bool.

Finally, we defined two **events** to store additional data. Their advantage with respect to mapping is that they are cheaper in terms of memory storage (and so gas usage).
The first event is focused on giving a card. This event takes the address of the player, its entity and the given card as arguments. The second event returns the winning entity, resets the counter and enables the player to start a new hand.



```
function GameBlackjack(uint256 costToPlay) payable public {
    uint256 PlayFee = msg.value;
}
```

This is a **constructor**, that is a **function** named after having defined the contract. A constructor for smart contract concerns how smart contracts are deployed. Here, the constructor has to be **payable** and **public** (payable is a method that allows to receive ETH when it is called, while public allows everyone to call such function). In this constructor we allow the player to choose the amount of ether to bet.



```
function StartPlaying() payable public {
    require(HandValueDealer[msg.sender] == 0);
    require(msg.value == PlayFee);
    require(msg.value <= (200000000 + 2 * block.gaslimit)/100000000);
    require(this.balance >= PlayFee * 5/2 * (OngoingGames + 1));
    OngoingGames++;

    giveCard(Subject.Gamer);
    giveCard(Subject.Dealer);
    giveCard(Subject.Gamer);

    if(HandValueGamer[msg.sender] == 11 && Aceforgamer[msg.sender]) {
        BlackjackforGamer[msg.sender] = true;
        EndGame(Subject.Gamer);
    } else {
        BlackjackforGamer[msg.sender] = false;
    }
}
```

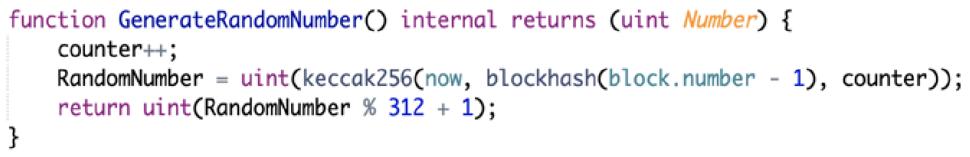
The function ***StartPlaying()*** allows to start a game. **Require** is like an assert. We want to be sure that:

- each player plays one hand at a time;
- the player sent the right amount of ETH that he/she previously defined;
- the bet placed by the player is lower than the expected block reward;
- the dealer/casino has enough money in its account to pay all potential winners. In particular, for every game in progress, the balance has to be greater than or equal to the sum of all the bets placed times 5/2 (the maximum payoff possible, if all players won with a Blackjack).

For what concerns the third requirement, we decided to approximate the block reward using the following formula:

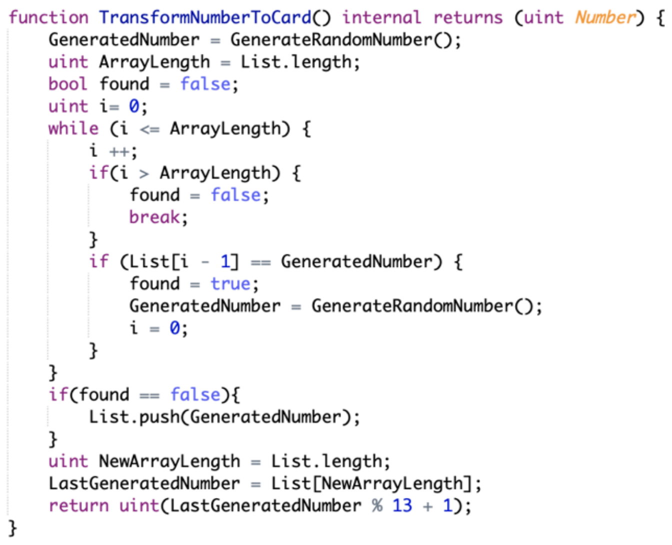
$$Block\ reward\ =\ static\ reward\ +\ block.gaslimit\ *\ (avg.gas\ price)\ +\ avg.n.uncles\ *\ 1/32\ *\ 2$$

The static reward considered is 2 (ETH), while for the gas price we took the average gas price for the last 100 days. We did not consider the average number of block uncles (and so the block reward coming from them) since we were not able to store reliable information about them.
If these conditions are not met, the function returns an error and the contract will not start the game. On the contrary, the game starts. The number of ongoing games is augmented by one and one card is given to the dealer, while two of them are delivered to the player (the second card of the dealer is not distributed yet for security reasons). Lastly, the function directly checks if the gamer has Blackjack and immediately assigns the win to the gamer if that is the case.



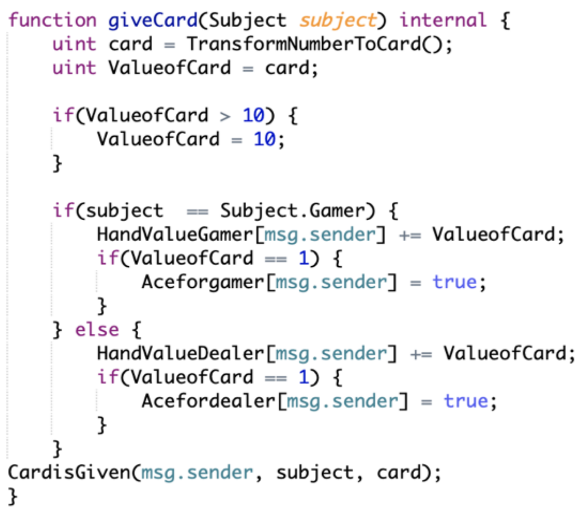
```
function GenerateRandomNumber() internal returns (uint Number) {
    counter++;
    RandomNumber = uint(keccak256(now, blockhash(block.number - 1), counter));
    return uint(RandomNumber % 312 + 1);
}
```

GenerateRandomNumber() generates a random number. It takes the value generated by hashing together the current timestamp, the hash of the previous block number and the value of the counter, and it returns a value that goes from 1 to 312 (the total number of cards in a typical deck used to play Blackjack).



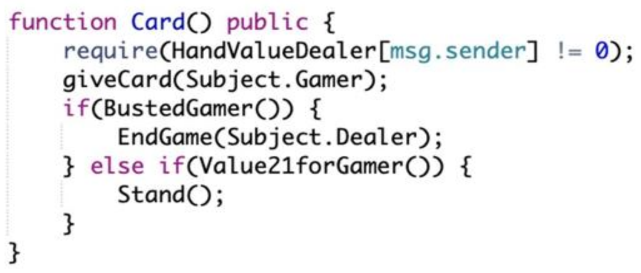
```
function TransformNumberToCard() internal returns (uint Number) {
    GeneratedNumber = GenerateRandomNumber();
    uint ArrayLength = List.length;
    bool found = false;
    uint i= 0;
    while (i <= ArrayLength) {
        i ++;
        if(i > ArrayLength) {
            found = false;
            break;
        }
        if (List[i - 1] == GeneratedNumber) {
            found = true;
            GeneratedNumber = GenerateRandomNumber();
            i = 0;
        }
    }
    if(found == false){
        List.push(GeneratedNumber);
    }
    uint NewArrayLength = List.length;
    LastGeneratedNumber = List[NewArrayLength];
    return uint(LastGeneratedNumber % 13 + 1);
}
```

The function ***TransformNumberToCard()*** checks if the generated number from the previous function is already in ***List***. *List* represents the collection of cards that have already been dealt. If the generated number is not in the list, the card is given to the dealer or player and added to the list. The goal of this function is to simulate the deck of cards available in a real blackjack game, which is composed by 6 decks of 52 cards each. Therefore, we check if any number between 1 to 312 has already been drawn to simulate that the same card cannot be dealt twice. In this way we manage to account for the fact that if the first card dealt is an ace, there are 23 of them in the deck left. As a consequence, the probability that the second card is an ace is slightly lower than the probability of dealing a card that has never come out yet (24/311 vs 23/311). At the end, the function transforms the last added number of the list into a value between 1 and 13 representing the 13 different cards of the same type in each deck: cards from ace to 10, Jack, Queen and King. The suit of a card has no relevance in blackjack, so we do not need to keep track of it.



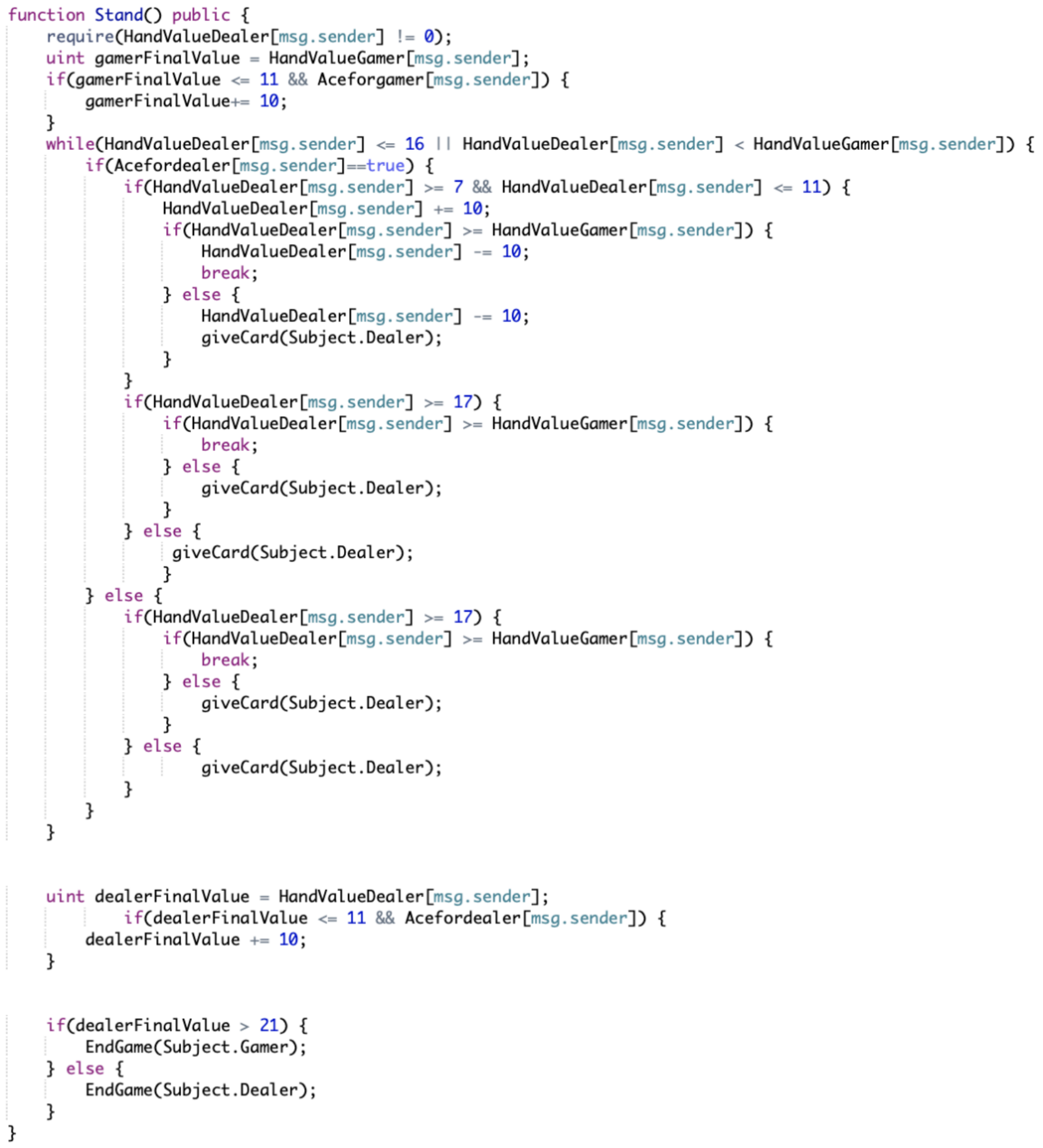
```
function giveCard(Subject subject) internal {
    uint card = TransformNumberToCard();
    uint ValueofCard = card;

    if(ValueofCard > 10) {
        ValueofCard = 10;
    }

    if(subject  == Subject.Gamer) {
        HandValueGamer[msg.sender] += ValueofCard;
        if(ValueofCard == 1) {
            Aceforgamer[msg.sender] = true;
        }
    } else {
        HandValueDealer[msg.sender] += ValueofCard;
        if(ValueofCard == 1) {
            Acefordealer[msg.sender] = true;
        }
    }
CardisGiven(msg.sender, subject, card);
}
```

The function ***giveCard()*** hits a card to the entity who asked for it. First of all, it generates a card using the previously defined function. If the value of the card is greater than 10 (so it is a Jack, a Queen or a King) we consider it as 10, otherwise we don't need any special treatment for the other cards (except for the ace). Once we determined the value of the card, the function adds it to the hand value of the entity.
It is important to know whether the hit card is an Ace because it can assume value 11 or 1 based on different conditions (we will explain this point later). Lastly, the function records the event.



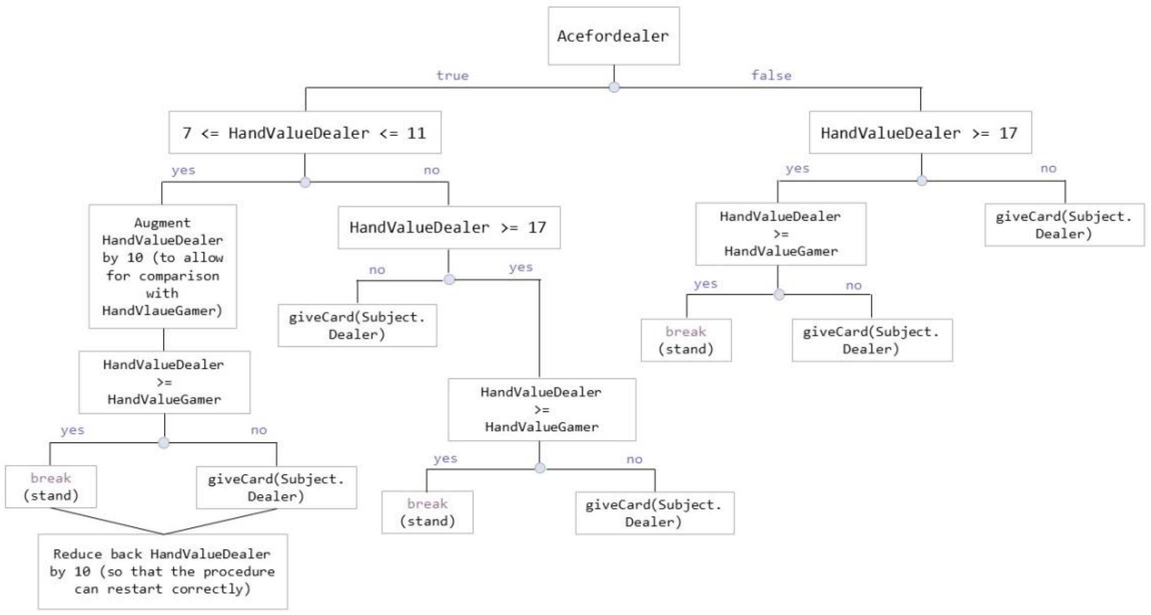
```
function Card() public {
    require(HandValueDealer[msg.sender] != 0);
    giveCard(Subject.Gamer);
    if(BustedGamer()) {
        EndGame(Subject.Dealer);
    } else if(Value21forGamer()) {
        Stand();
    }
}
```

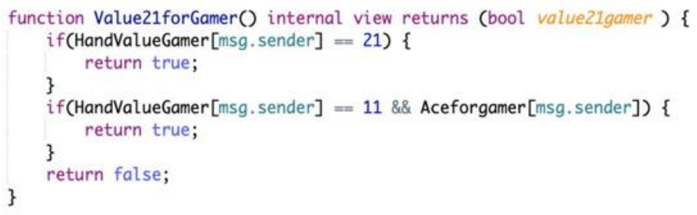
The function ***Card()*** determines the conditions to give a card or to end the game. The function requires that a game is in progress (dealer hand value is different from zero). If the condition is met, the card is given to the player. If the gamer has more than 21 the game is over, otherwise if the player has 21 he/she stands.



```
function Stand() public {
    require(HandValueDealer[msg.sender] != 0);
    uint gamerFinalValue = HandValueGamer[msg.sender];
    if(gamerFinalValue <= 11 && Aceforgamer[msg.sender]) {
        gamerFinalValue+= 10;
    }
    while(HandValueDealer[msg.sender] <= 16 || HandValueDealer[msg.sender] < HandValueGamer[msg.sender]) {
        if(Acefordealer[msg.sender]==true) {
            if(HandValueDealer[msg.sender] >= 7 && HandValueDealer[msg.sender] <= 11) {
                HandValueDealer[msg.sender] += 10;
                if(HandValueDealer[msg.sender] >= HandValueGamer[msg.sender]) {
                    HandValueDealer[msg.sender] -= 10;
                    break;
                } else {
                    HandValueDealer[msg.sender] -= 10;
                    giveCard(Subject.Dealer);
                }
            }
            if(HandValueDealer[msg.sender] >= 17) {
                if(HandValueDealer[msg.sender] >= HandValueGamer[msg.sender]) {
                    break;
                } else {
                    giveCard(Subject.Dealer);
                }
            } else {
                 giveCard(Subject.Dealer);
                }
        } else {
            if(HandValueDealer[msg.sender] >= 17) {
                if(HandValueDealer[msg.sender] >= HandValueGamer[msg.sender]) {
                    break;
                } else {
                    giveCard(Subject.Dealer);
                }
            } else {
                    giveCard(Subject.Dealer);
            }
        }
    }

    uint dealerFinalValue = HandValueDealer[msg.sender];
            if(dealerFinalValue <= 11 && Acefordealer[msg.sender]) {
        dealerFinalValue += 10;
    }

    if(dealerFinalValue > 21) {
        EndGame(Subject.Gamer);
    } else {
        EndGame(Subject.Dealer);
    }
}
```

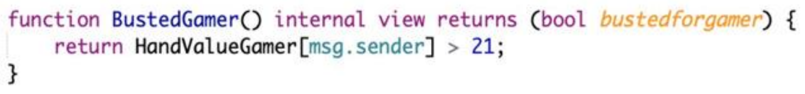
The function ***Stand()*** is the core constructor that governs most of the game dynamics and the strategies of the dealer. After initially checking that a game is in progress, it calculates the final score value of the gamer, taking into account the possibility that he/she is using the ace as 11 instead of 1. Once the gamer has finished, it is now the moment for the dealer to try to beat him. One blackjack rule is fundamental at this point: the dealer has to reach the minimum value of 17, therefore a card is automatically given until this value is reached. As stated earlier though, the ace can take the value of 1 or 11 depending on what is convenient for the hand at play. This complicates quite a bit the decision process and therefore the function's structure is summarized in the diagram below to have a nicer and more understandable representation of the dynamics. The ultimate goal of the dealer, once it has reached 17, is to either beat the gamer or get busted trying. This is slightly different to what happens on a standard blackjack table at a casino: being a one-on-one game there is no reason for the dealer to play conservative. At the end, we check whether the dealer has a value greater than 21 or if it has beaten the gamer and we call the function ***EndGame*** specifying the respective winner.


Acefordealer
true
false
7 <= HandValueDealer <= 11
yes
no
HandValueDealer >= 17
yes
no
Augment HandValueDealer by 10 (to allow for comparison with HandVlaueGamer)
HandValueDealer >= 17
no
yes
giveCard(Subject. Dealer)
HandValueDealer >= HandValueGamer
yes
no
break (stand)
giveCard(Subject. Dealer)
Reduce back HandValueDealer by 10 (so that the procedure can restart correctly)
HandValueDealer >= HandValueGamer
yes
no
break (stand)
giveCard(Subject. Dealer)
HandValueDealer >= HandValueGamer
yes
no
break (stand)
giveCard(Subject. Dealer)
giveCard(Subject. Dealer)



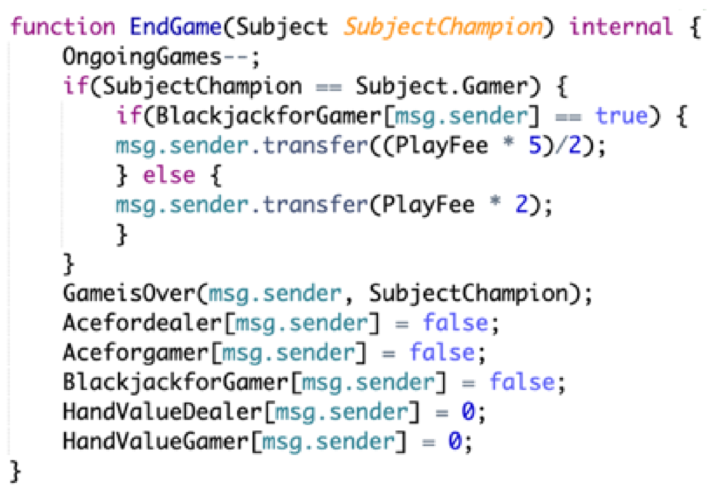
```
function Value21forGamer() internal view returns (bool value21gamer ) {
    if(HandValueGamer[msg.sender] == 21) {
        return true;
    }
    if(HandValueGamer[msg.sender] == 11 && Aceforgamer[msg.sender]) {
        return true;
    }
    return false;
}
```

The function ***Value21Gamer()*** accounts for the situation in which the player has 21. If this happens, he/she stands automatically.



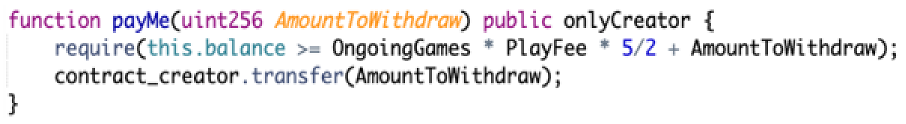
```
function BustedGamer() internal view returns (bool bustedforgamer) {
    return HandValueGamer[msg.sender] > 21;
}
```

The function ***BustedGamer()*** represents the situation in which the player scores more than 21, so he/she automatically loses the game.



```
function EndGame(Subject SubjectChampion) internal {
    OngoingGames--;
    if(SubjectChampion == Subject.Gamer) {
        if(BlackjackforGamer[msg.sender] == true) {
        msg.sender.transfer((PlayFee * 5)/2);
        } else {
        msg.sender.transfer(PlayFee * 2);
        }
    }
    GameisOver(msg.sender, SubjectChampion);
    Acefordealer[msg.sender] = false;
    Aceforgamer[msg.sender] = false;
    BlackjackforGamer[msg.sender] = false;
    HandValueDealer[msg.sender] = 0;
    HandValueGamer[msg.sender] = 0;
}
```

The function ***EndGame()*** determines who is the winner. According to the winning entity, the dealer pays the player or keeps the money. In case of simple winning, the player will receive 2 times the initial placed bet. In case of Blackjack, the player will receive 5/2 times the initial bet. Finally, the game is over, so we reset the values of variables linked to the player address and we lower the number of ongoing games.



```
function payMe(uint256 AmountToWithdraw) public onlyCreator {
    require(this.balance >= OngoingGames * PlayFee * 5/2 + AmountToWithdraw);
    contract_creator.transfer(AmountToWithdraw);
}
```

The function ***payMe()*** allows the owner of the contract to specify how much to withdraw. The requirement ensures that the dealer cannot withdraw if it does not have enough money to pay in case each player wins its game.

The RNG problem

The most important problem we faced when we built this smart contract concerned the generation of a truly random number. What we thought initially was to take as inputs the player's address and the hash of a new (not already) mined block. Then, once the gamer presses the button 'get a new card', the block hash of this newly mined block, together with the player's address, will be hashed together to form a new hash that will be used to calculate which card will be distributed. We found out that we were not able to implement such RNG because of limitations of Solidity. In fact, the native function ***blockhash()*** is able to give us the hash of the previously defined block (not even the current one), so we have to stick with this limitation. For this reason, we came out with the **RNG method** previously defined since it is one of the most used one in the Solidity community.
Even if we did not take the hash of the next mined block, there still can be the possibility that by using this system, however, a player who has a huge computing power mines a valid block and sees in advance the card that will be distributed to him or to the dealer. Once he/she saw it, he/she could decide to broadcast the block to the network (or not), giving him/her the possibility to cheat. This situation could be avoided by setting the maximum amount of ETH a player/croupier could bet to be less than the average block reward.

Conclusion

In this report we dove deep into the technical side of our project, explaining how we implemented the smart contract for playing blackjack on the Ethereum blockchain. We also listed the hypothesis we made, as well as the limitations we had to stick with.
In the same GitHub repository reported before, you can find a PowerPoint presentation where we explain the business part of our project, as well as a simulation of the Blackjack.

References

- https://solidity.readthedocs.io/en/
- http://remix.ethereum.org/
- https://media.consensys.net/the-thirdening-what-you-need-to-know-df96599ad857
- https://ethereum.stackexchange.com/questions/5958/how-to-query-the-amount-of-mining-reward-from-a-certain-block
- https://etherscan.io/chart/gasprice (download csv and average of previous 100 days)
- https://github.com/Ziobero/Blockchain_based_blackjack