Design Patterns Decisions

Useful design patterns were used in the creation of this smart contract.

Contract Self Destruction

Self-destructing a smart contract is the process of removing it from the blockchain altogether. After self-destruction, it will be no longer possible to perform operations on the smart contract.

In the roulette smart contract, the table is closed by the destruction of the smart contract. Only the croupier is allowed to call this function. Any remaining funds will be transferred to him/her.

This is an inefficient way for replayability. A less expensive approach is to reset any values holding game data. A more elaborate approach is to use data stores to hold game session identifiers, along with players and game data linked to said game session (source).

```
function closeTable() public ownerRequired {
    require(!isTableOpen,"The table is not closed for betting.");
    require(winningNumber > -1, "No winning number is set.");
    // Convert croupier from address to address payable.
    selfdestruct(address(uint160(croupier)));
}
```

Mapping Iterator

In Solidity, the mapping data type is very useful but cannot be iterated. The mapping iterator pattern is the process of using an array holding the keys of the mapping variable and iterate through that instead (source).

```
// Array of players participating in this table
address[] private players;

// Mapping to check if an address placed a bet on either odd or even numbers.
mapping(address => PlayerSession) private playerSessions;

// Sets the number drawn and assigns winnings.
function setWinningNumber(int16 numberDrawn) public ownerRequired {

    // Code removed for brevity

    // Iterate through all the players registered with this table.
    // Mapping Iterator pattern.
    for(uint i = 0; i < players.length; i++){
        // Get the session for this table
        PlayerSession memory playerSession = playerSessions[players[i]];
        uint totalWinAmount = 0;

        // Process Odd / Even bets; with 1:1 winning ratio.
```

```
        if (playerSession.oddEvenBet == winOddEven)
        {
            totalWinAmount += (tableStakeInWei * 2);
        }

        // At the end of it, all map the winnings
        playerWinnings[players[i]] = totalWinAmount;
    }
}
```

Withdrawal Pattern

Sending funds to other contracts may impose several unnecessary risks to the sender. The recipient might purposely throw exceptions in its fallback function in order to deplete the funds of its callers or the sender might run out of gas. When paying multiple recipients, the process might crash halfway through the list, allowing the method to be called again or not having the paid recipient list properly updated. This can allow a recipient to be paid indefinitely until the sender's funds are depleted.

A safer approach is to create a function, allowing users to withdraw funds. In the developed system, players who are eligible for withdrawal will have Wei transferred to their address (source).

```
// Winnings are not paid automatically, instead the users call the withdraw
function.
// Reference: Mastermind.sol > withdrawWinnings()
// This is a security feature to avoid exploits from bugs, also referred to as the
Withdrawal pattern.
function withdrawWinnings() public {
    require(!isTableOpen,"The table is not closed for betting.");
    require(winningNumber > -1, "No winning number is set.");
    require(playerWinnings[msg.sender] != 0, "There are no withdrawable
winnings.");
    require(address(this).balance >= playerWinnings[msg.sender], "There are not
enough funds to transfer.");
    uint transferAmount = playerWinnings[msg.sender];

    // Set the winnings mapping to 0 before the transfer.
    playerWinnings[msg.sender] = 0;
    require(msg.sender.send(transferAmount));
}
```