```
pragma solidity =0.4.24;
import "https://github.com/smartcontractkit/chainlink/evm/contracts/ChainlinkClient.sol";
contract DSSafeAddSub {
    function safeToAdd(uint a, uint b) internal pure returns (bool) {
        return (a + b >= a);
    }
    function safeAdd(uint a, uint b) internal pure returns (uint) {
        if (!safeToAdd(a, b)) revert();
        return a + b;
    }

    function safeToSubtract(uint a, uint b) internal pure returns (bool) {
        return (b <= a);
    }

    function safeSub(uint a, uint b) internal pure returns (uint) {
        if (!safeToSubtract(a, b)) revert();
        return a - b;
    }
    function safeMul(uint _a, uint _b) internal pure returns (uint) {
        uint c = _a * _b;
        require(_a==0 || c/_a==_b);
        return c;
    }
}

contract CeeLoo is DSSafeAddSub,ChainlinkClient{

    // A structure representing a single bet.

    struct Bet {
        uint amount;
        uint winamount;
        uint8 modulo1;
        uint8 modulo2;
        uint8 modulo3;
        uint40 placeBlockNumber;
        address gambler;
        uint lockedAmount;
    }
    address public owner;
    address[] public players;
```

```
    uint public playerPartTime;
    uint public bankerRollTime;
    uint public playerRollTime;
    mapping(address => Bet) public bets;
    mapping (address => uint) playerProfit;
    uint8 public setpointByBanker;
    bool public bankerBetPlaced;
    bool public gamePaused;
    uint public betValue1 = 0.01 ether; // these are the amount againt which player can put their
bet
    uint public betValue2 = 0.02 ether;
    uint public betValue3 = 0.03 ether;
    uint public betValue4 = 0.05 ether;
    uint8 private dice1 = 1;
    uint8 private dice2 = 2;
    uint8 private dice3 = 3;
    uint8 private dice4 = 4;
    uint8 private dice5 = 5;
    uint8 private dice6 = 6;
    uint256 private randomNumber1;
    uint256 private randomNumber2;
    uint256 private randomNumber3;

    // Adjustable max bet profit. Used to cap bets against dynamic odds.

    uint public maxProfit = 0.05 ether;

    // Funds that are locked in potentially winning bets. Prevents contract from
    // committing to bets it cannot pay out.
    uint public lockedInBets;
    uint constant HOUSE_EDGE_PERCENT = 1;
    bool public payoutsPaused;

    // Chainlink veriables
    //   Rinkeby
    // LINK token address: 0x01BE23585060835E02B77ef475b0Cc51aA1e0709
    // Oracle address: 0x7AFe1118Ea78C1eae84ca8feE5C65Bc76CcF879e
    // JobID: 0xb00ed7210563488cbe5a3b7729c0ec72

    uint256 constant internal LINK = 10**18;
    uint256 constant internal ORACLEPAYMENT = 1 * LINK;
    address public ORACLE;
    bytes32 public jobID;
```

```
    address public LINK_TOKEN;

    // General events

    event FailedPayment(address indexed beneficiary, uint amount);
    event Payment(address indexed beneficiary, uint amount, bool status, string msg);
    event Participate(address indexed player, uint amount, string msg, bool status, uint
timestamp);
    event BankerWin(string msg, uint8 num1, uint8 num2, uint8 num3);
    event setPoint(string msg, uint);
    event tryAgainOwner(string msg);
    event PlayerWin(string msg, uint winningamount, address addr, uint8 num1, uint8 num2,
uint8 num3);
    event playerRollDice(string msg);
    event LogOwnerTransfer(address addr, uint value);
    event BankerLoose(string msg);

    // chainLink function for random number generation

    function getRandom() onlyOwner external {
        Chainlink.Request memory req1 = buildChainlinkRequest(jobID, this, this.fulfill1.selector);
        Chainlink.Request memory req2 = buildChainlinkRequest(jobID, this, this.fulfill2.selector);
        Chainlink.Request memory req3 = buildChainlinkRequest(jobID, this, this.fulfill3.selector);
        req1.addUint("min", 1);
        req1.addUint("max", 1000000000);
        sendChainlinkRequestTo(ORACLE, req1, ORACLEPAYMENT);
        req2.addUint("min", 1);
        req2.addUint("max", 1000000000);
        sendChainlinkRequestTo(ORACLE, req2, ORACLEPAYMENT);
        req3.addUint("min", 1);
        req3.addUint("max", 1000000000);
        sendChainlinkRequestTo(ORACLE, req3, ORACLEPAYMENT);
    }

    function fulfill1(bytes32 _requestId, uint8 _data)
        public
        recordChainlinkFulfillment(_requestId)
    {
        randomNumber1 = _data;
    }

    function fulfill2(bytes32 _requestId, uint8 _data)
        public
```

```
    recordChainlinkFulfillment(_requestId)
{
    randomNumber2 = _data;
}

function fulfill3(bytes32 _requestId, uint8 _data)
    public
    recordChainlinkFulfillment(_requestId)
{
    randomNumber3 = _data;
}

constructor(address token, address oracle, bytes32 jobId) public payable {
    require(token != address(0));
    require(oracle != address(0));
    require(jobId != 0);
    owner = msg.sender;
    gamePaused = false;
    bankerBetPlaced = false;
    ORACLE = oracle;
    LINK_TOKEN =token;
    jobID = jobId;
}

modifier onlyOwner {
    if (msg.sender != owner) revert();
    _;
}

/* only owner address can set emergency pause #1 */

function ownerPauseGame(bool newStatus) public onlyOwner
{
        gamePaused = newStatus;
}

/*
 * checks game is currently active
*/
modifier gameIsActive {
    if(gamePaused == true) revert();
    _;
}
```

```
    // check contract balance

    function balContract() public view returns (uint){
        uint bal = address(this).balance;
        return bal ;
    }
    // ownership transfer
    function ownerTransferOwnership(address newOwner) public onlyOwner{
        owner = newOwner;
    }

    // Change max bet reward. Setting this to zero effectively disables betting.

    function setMaxProfit(uint _maxProfit) public onlyOwner {
        require (_maxProfit < betValue4, "maxProfit should be a sane number.");
        maxProfit = _maxProfit;
    }

    // Contract may be destroyed only when there are no ongoing bets,
    // either settled or refunded. All funds are transferred to contract owner.

    function kill() external onlyOwner {
        require (lockedInBets == 0, "All bets should be processed (settled or refunded) before
self-destruct.");
        selfdestruct(address(uint160(owner)));
    }

    // contract accept ether

    function() external payable {

    }

    //auto win condition1, if rolls 4,5,6

    function autowin1(uint8 num1, uint8 num2, uint8 num3)  private view returns (bool){
          if(num1 == dice4 || num2 == dice4 || num3 == dice4){
           if(num1 == dice5 || num2 == dice5 || num3 == dice5){
              if(num1 == dice6 || num2 == dice6 || num3 == dice6){
                 return true;
              }else {
                 return false;
```

```
            }
        }else {
            return false;
        }
    }else{
        return false;
    }
}

//auto win condition2, if user throw 6 with double

function autowin2(uint8 num1, uint8 num2, uint8 num3) private pure returns (bool){
    if(num1== 6 || num2 == 6 || num3 == 6){
        if(num1 == num2 || num1 == num3 || num2 == num3){
            return true  ;
        }else{
            return false;
        }
    }else {
        return false ;
    }
}

//auto win condition3, user throw tripple

function autowin3(uint8 num1, uint8 num2, uint8 num3)  private pure returns (bool) {
    if(num1 == num2 && num2 == num3){
        return true;
    }
    else{
        return false;
    }

}

// auto loose condition1, if rolls 1,2,3

function autoloos1(uint8 num1, uint8 num2, uint8 num3) private view returns (bool){
    if(num1 == dice1 || num2 == dice1 || num3 == dice1){
        if(num1 == dice2 || num2 == dice2 || num3 == dice2){
            if(num1 == dice3 || num2 == dice3 || num3 == dice3){
                return true;
            }else{
```

```
            return false;
          }
       }else{
          return false;
       }
    }else{
       return false;
    }
}

// auto loose condition2,double with 1

function autoloos2(uint8 num1, uint8 num2, uint8 num3) private pure returns (bool){
    if(num1 == 1 || num2 == 1 || num3 == 1){
       if(num1 == num2 || num1 == num3 || num2 == num3){
          return true  ;
       }else{
          return false;
       }
    }else{
       return false ;
    }
}

// set point role a pair with single(exept 1,6)

function setPoint1(uint8 num1, uint8 num2, uint8 num3) private pure returns (uint8){
    if(num1 == num2){
       if(num3 == 2 || num3 == 3 || num3 == 4 || num3 == 5){
          return num3;
       }else{
          return 0;
       }
    }else if(num2 == num3){
       if(num1 == 2 || num1 == 3 || num1 == 4 || num1 == 5){
          return num1;
       }else{
          return 0;
       }
    }else if(num1 == num3){
       if(num2 == 2 || num2 == 3 || num2 == 4 || num2 == 5){
          return num2;
       }else{
```

```
            return 0;
        }
    }else{
        return 0;
    }
}

// check valid input ether value

function validAmount(uint amount) internal view returns (bool){
    if(amount != betValue1 && amount != betValue2 && amount !=betValue3 && amount
!=betValue4){
        return false ;
    }else{
        return true ;
    }
}

function contractBalance() public view returns (uint256){
    uint256 bal = address(this).balance;
    return bal ;
}

// check whether contract could pay further winning amount

function contractBal(uint amount) internal view returns (bool){
    uint conBal = address(this).balance;
    uint lockedBets = safeAdd(lockedInBets, amount);
    uint lockedBal = safeMul(2,lockedBets);
    if(conBal >= lockedBal){
        return true;
    }else{
        return false;
    }
}

function placeBetBanker() onlyOwner external {

    // Check that the bet is in 'clean' state

    require(!bankerBetPlaced, "Bet has been placed already");

    //banker can only place the bet after players participated in current game
```

```
        require(players.length >=1 && players.length <= 10, "min palyer 1 and max 10");

        // Banker can place the bet after players participated in current game

        require(block.number > playerPartTime);
        //banker will auto loose if he takes more than 15 block to place his bet.

        if(bankerRollTime <= block.number){
            for (i =0; i < players.length; i++){
                addr = players[i];
                bet = bets[addr];
                value = bet.amount;
                require (value != 0, "Bet should be in an 'active' state"); // Check that bet is in 'active'
state.
                uint winAmount = getDiceWinAmount(value);// winning amount.
                bet.winamount += safeAdd(winAmount,value);
                bet.amount = 0;
                bet.placeBlockNumber = uint40(block.number);
            }
            delete players;
            emit BankerWin("Banker has lost the game due to time out", modulo1, modulo2,
modulo3);
            bankerBetPlaced = true;
            return;
        }
        uint8 modulo1 =
uint8(uint256(keccak256(abi.encodePacked(randomNumber1,randomNumber2,randomNumber
3,msg.sender, block.timestamp))) % 5 + 1);
        uint8 modulo2 =
uint8(uint256(keccak256(abi.encodePacked(randomNumber2,randomNumber3,
randomNumber1, msg.sender, block.timestamp))) % 5 + 1);
        uint8 modulo3 =
uint8(uint256(keccak256(abi.encodePacked(randomNumber3,randomNumber1,randomNumber
2, msg.sender, block.timestamp))) % 5 + 1);

        address addr;
        uint i = 0;
        uint value = 0;

        //Banker auto win condition , throws 4,5,6

        if(autowin1(modulo1,modulo2,modulo3)){
```

```
    for (i =0; i < players.length; i++){
        addr = players[i];
        Bet storage bet = bets[addr];
        value = bet.amount;
        // Move bet into 'processed' state already.
        bet.amount = 0;
        // bet.lockedAmount = 0;
        // bet.gambler = 0x0;
        bet.lockedAmount = safeSub(bet.lockedAmount, value);
        lockedInBets = safeSub(lockedInBets, value);
    }
    delete players;
    bankerBetPlaced = true;
    emit BankerWin("banker has won the game 1", modulo1, modulo2, modulo3);
    return ;
}

// banker throws tripple

if (autowin3(modulo1, modulo2, modulo3)){
    for (i =0; i < players.length; i++){
        addr = players[i];
        bet = bets[addr];
        value = bet.amount;
        bet.amount = 0;     // Move bet into 'processed' state already.
        bet.lockedAmount = safeSub(bet.lockedAmount, value);
        lockedInBets = safeSub(lockedInBets, value);
    }
    delete players;
    emit BankerWin("banker has won the game 2", modulo1, modulo2, modulo3);
    bankerBetPlaced = true;
    return ;
}

// banker throws double with 6

if(autowin2(modulo1, modulo2, modulo3)){
    for (i=0; i < players.length; i++) {
        addr = players[i];
        bet = bets[addr];
        value = bet.amount;
        bet.amount = 0;
        bet.lockedAmount = safeSub(bet.lockedAmount, value);
```

```
            lockedInBets = safeSub(lockedInBets, value);
        }
        delete players;
        emit BankerWin("banker has won the game 3", modulo1, modulo2, modulo3);
        bankerBetPlaced = true;
        return;
    }

    //setpoint by Banker againt which players can bet

    uint8 point = setPoint1(modulo1, modulo2, modulo3);
    if (point != 0){
        setpointByBanker = point;

        //this is the time given to all players to roll the dice after than player will auto loose the
game

        playerRollTime = safeAdd(block.number, 15);
        delete players;
        bankerBetPlaced = true;
        emit setPoint("player will bet against the set point",point);
        return;
    }

    //Banker auto loose if rolls 1,2,3

    if(autoloos1(modulo1, modulo2, modulo3)){
        for (i =0; i < players.length; i++){
            addr = players[i];
            bet = bets[addr];
            value = bet.amount;
            require (value != 0, "Bet should be in an 'active' state");   // Check that bet is in 'active'
state.
            winAmount = getDiceWinAmount(value);  // winning amount.
            bet.winamount += safeAdd(winAmount,value);
            bet.amount = 0;
            bet.placeBlockNumber = uint40(block.number);
        }
        delete players;
        emit BankerWin("Banker has lost the game1", modulo1, modulo2, modulo3);
        bankerBetPlaced = true;
        return;
    }
```

```
        //Banker auto loose if rolls double with 1

        if(autoloos2(modulo1, modulo2, modulo3)){
            for (i = 0; i < players.length; i++){
                addr = players[i];
                bet = bets[addr];
                value = bet.amount;
                require (value != 0, "Bet should be in an 'active' state"); // Check that bet is in 'active'
state.
                winAmount = getDiceWinAmount(value);    // winning amount
                bet.winamount += safeAdd(winAmount,value);
                bet.amount = 0;
                bet.placeBlockNumber = uint40(block.number);
            }
            delete players;
            bankerBetPlaced = true;
            emit BankerWin("banker has lost the game2", modulo1, modulo2, modulo3);
            return ;
        }

        emit BankerWin("will roll again the dice", modulo1, modulo2, modulo3);
        bankerBetPlaced = false;
        return;
    }

    // 1st player need to Participate with a fixed amount of ether

    function playerParticipate() gameIsActive external payable returns (string, bool, uint) {
        require(players.length <= 10, "min palyer 1 and max 10");
        // Check that the bets is in 'clean' state.
        if(players.length  == 0){
            require(msg.sender != owner);
            // time till all Player can join the game in current round
            Bet storage bet = bets[msg.sender];
            // Validate input data ranges.
            uint amount = msg.value;
            require(bet.amount == 0,"Bet should be in a 'clean' state."); //bet.gambler == address(0)
&&
            // guard againt invalid value;
            require(validAmount(amount),"amount should be in range");
            //contract should he enough balance to payout all the bets placed by all players
            require(contractBal(amount), "insufficient amount inside contract");
```

```
            playerPartTime = safeAdd(block.number,10);
            bankerRollTime = safeAdd(block.number, 25);
            lockedInBets += amount;
            bet.amount = msg.value;
            bet.lockedAmount += msg.value;
            bet.gambler = msg.sender;
            players.push(msg.sender);
            bankerBetPlaced = false;
            emit Participate(msg.sender, msg.value, "bet placed successfully", true, block.number);
            return ("successs", true, block.number);
        }

        // current block should be less than player participation time in current game

        if(block.number <=  playerPartTime){
            bet = bets[msg.sender];
            // Validate input data ranges.
            amount = msg.value;
            require(msg.sender != owner);
            require(bet.amount == 0,"Bet should be in a 'clean' state."); //bet.gambler == address(0)
&&
            // guard againt invalid value;
            require(validAmount(amount),"amount should be in range");
            //contract should he enough balance to payout all the bets placed by all players
            require(contractBal(amount), "insufficient amount inside contract");
            lockedInBets += uint(amount);
            bet.amount = msg.value;
            bet.gambler = msg.sender;
            bet.lockedAmount += msg.value;
            players.push(msg.sender);
            emit Participate(msg.sender, msg.value, "bet placed successfully", true, block.number);
            return ("successs", true, block.number);
        }
        return ("please wait for next round", false, block.number);
        emit Participate(msg.sender, msg.value, "please wait for next round", false, block.number);
    }

    // *** Betting logic

    // Bet states:
    //  amount == 0 && gambler == 0 - 'clean' (can place a bet)
    //  amount != 0 && gambler != 0 - 'active' (can be settled or refunded)
    //  amount == 0 && gambler != 0 - 'processed' (can clean storage)
```

```
    //  NOTE: Storage cleaning is not implemented in this contract version; it will be added
    // Bet placing transaction - issued by the player.

    function diceRoll() gameIsActive external {

        // after banker player can roll the dice

        require(bankerBetPlaced, "first banker will place the bet");
        require(msg.sender != owner);
        // Check that the bet is in 'clean' state.
        Bet storage bet = bets[msg.sender];
        require(bankerBetPlaced, "after banker placing the bet player can bet");
        require(bet.amount != 0, "1st need to Participate"); //bet.gambler != address(0) &&
        uint8 modulo1 =
uint8(uint256(keccak256(abi.encodePacked(randomNumber1,randomNumber2,randomNumber
3, msg.sender, block.timestamp))) % 5 + 1);
        uint8 modulo2 =
uint8(uint256(keccak256(abi.encodePacked(randomNumber2,randomNumber3,
randomNumber1, msg.sender, block.timestamp))) % 5 + 1);
        uint8 modulo3 =
uint8(uint256(keccak256(abi.encodePacked(randomNumber3,randomNumber1,randomNumber
2, msg.sender, block.timestamp))) % 5 + 1);

        uint amount = 0;
        uint winAmount = 0;

        // if he takes more than 15 blocks to roll the dice, player get auto loose;

        if(playerRollTime < block.number){

            // Store bet parameters on blockchain

            lockedInBets = safeSub(lockedInBets, bet.amount);
            bet.modulo1 = modulo1;
            bet.modulo2 = modulo2;
            bet.modulo3 = modulo3;
            bet.lockedAmount = safeSub(bet.lockedAmount, bet.amount);
            bet.amount= 0;
            bet.placeBlockNumber = uint40(block.number);
            emit PlayerWin("player loose the game due to time expire",0,bet.gambler, modulo1,
modulo2, modulo3);
            return;
        }
```

```
        //Player throws 4,5,6

        if (autowin1(modulo1, modulo2,modulo3)){

            // Winning amount.

            amount = bet.amount;
            winAmount = getDiceWinAmount(amount);

            // Enforce max profit limit.
            require (winAmount <= amount + maxProfit, "maxProfit limit violation.");

            // Store bet parameters on blockchain.
            bet.winamount += safeAdd(winAmount,amount);
            bet.amount = 0;
            bet.modulo1 = modulo1;
            bet.modulo2 = modulo2;
            bet.modulo3 = modulo3;
            bet.placeBlockNumber = uint40(block.number);
            emit PlayerWin("player won the game",winAmount,bet.gambler, modulo1, modulo2,
modulo3);
            return;

        }

        // palyer roll tripple
        if (autowin3(modulo1, modulo2, modulo3)){
            amount = bet.amount; // winning amount.
            winAmount = getDiceWinAmount(amount);

            // Enforce max profit limit.
            require(winAmount <= amount + maxProfit, "maxProfit limit violation.");

            // store bet parameters on blockchain.

            bet.winamount += safeAdd(winAmount,amount);
            bet.amount = 0;
            // bet.gambler = 0x0;
            bet.modulo1 = modulo1;
            bet.modulo2 = modulo2;
            bet.modulo3 = modulo3;
            bet.placeBlockNumber = uint40(block.number);
```

```
            emit PlayerWin("player won the game",winAmount,bet.gambler, modulo1, modulo2,
modulo3);
            return;
        }

        // player throws double with 6

        if(autowin2(modulo1, modulo2, modulo3)){
            amount = bet.amount;
            winAmount = getDiceWinAmount(amount); // winning amount.

            // Enforce max profit limit.
            require (winAmount <= amount + maxProfit, "maxProfit limit violation.");

            // Store bet parameters on blockchain.
            bet.winamount += safeAdd(winAmount,amount);
            bet.amount = 0;
            bet.modulo1 = modulo1;
            bet.modulo2 = modulo2;
            bet.modulo3 = modulo3;
            bet.placeBlockNumber = uint40(block.number);
            emit PlayerWin("player won the game",winAmount,bet.gambler, modulo1, modulo2,
modulo3);
            return;
        }

        //setpoint by player

        uint8 point = setPoint1(modulo1, modulo2, modulo3);

        // if player win the game
        if (point > setpointByBanker){
            // player won the game
            amount = bet.amount;
            winAmount = getDiceWinAmount(amount); // winning amount.

            // Enforce max profit limit.
            require (winAmount <= amount + maxProfit, "maxProfit limit violation.");

            // Store bet parameters on blockchain.
            bet.winamount += safeAdd(winAmount,amount);
            bet.amount = 0;
            bet.modulo1 = modulo1;
```

```
            bet.modulo2 = modulo2;
            bet.modulo3 = modulo3;
            bet.placeBlockNumber = uint40(block.number);
            emit PlayerWin("player won the game",winAmount,bet.gambler, modulo1, modulo2,
modulo3);
            return;

        }else if(point < setpointByBanker){   // player loose the game

            // Store bet parameters on blockchain

            lockedInBets = safeSub(lockedInBets, bet.amount);
            bet.modulo1 = modulo1;
            bet.modulo2 = modulo2;
            bet.modulo3 = modulo3;
            bet.lockedAmount = safeSub(bet.lockedAmount, bet.amount);
            bet.amount= 0;
            bet.placeBlockNumber = uint40(block.number);
            emit PlayerWin("player loose the game",0,bet.gambler, modulo1, modulo2, modulo3);
            return;
        }else if(point == setpointByBanker){

            //game has been draw

            lockedInBets = safeSub(lockedInBets, bet.amount);
            // Store bet parameters on blockchain.
            bet.winamount = bet.amount;
            bet.modulo1 = modulo1;
            bet.modulo2 = modulo2;
            bet.modulo3 = modulo3;
            bet.lockedAmount = safeSub(bet.lockedAmount, bet.amount);
            bet.amount = 0;
            // bet.gambler = 0x0;
            bet.placeBlockNumber = uint40(block.number);
            emit PlayerWin("game ties",0,bet.gambler, modulo1, modulo2, modulo3);
            return;
        }

        //Player auto loose : 1,2,3
        if(autoloos1(modulo1, modulo2, modulo3)){
            // Store bet parameters on blockchain
            lockedInBets = safeSub(lockedInBets, bet.amount);
            bet.amount= 0;
```

```
            bet.modulo1 = modulo1;
            bet.modulo2 = modulo2;
            bet.modulo3 = modulo3;
            bet.lockedAmount = safeSub(bet.lockedAmount, bet.amount);
            bet.amount = 0;
            bet.placeBlockNumber = uint40(block.number);
            emit PlayerWin("player loose the game",0,bet.gambler, modulo1, modulo2, modulo3);
            return;
        }

        //Player loose, double with 1
        if(autoloos2(modulo1, modulo2, modulo3)){
            bet.lockedAmount = safeSub(bet.lockedAmount, bet.amount);
            // Store bet parameters on blockchain
            bet.modulo1 = modulo1;
            bet.modulo2 = modulo2;
            bet.modulo3 = modulo3;
            lockedInBets = safeSub(lockedInBets, bet.amount);
            bet.amount= 0;
            bet.placeBlockNumber = uint40(block.number);
            emit PlayerWin("player loose the game",0,bet.gambler, modulo1, modulo2, modulo3);
            return;
        }
        emit PlayerWin("player will roll again", 0,bet.gambler, modulo1, modulo2, modulo3);
        return;
    }


    // Common settlement code for settleBet.

    function settleBetCommon(address benificary) external returns(address, uint) {

        // fetch bet parameters into local variables (to save gas).

        require(benificary != address(0));
        Bet storage bet = bets[benificary];
        if(bet.winamount == 0 && bet.amount != 0){
            require(block.number > safeAdd(bet.placeBlockNumber,25), "if somehow banker do not
place the bet and players funds trap in contract");
            address gambler = bet.gambler;
            uint amount = bet.amount;
            bet.amount = 0; //move bet into 'processed' state already.
            // unlock the bet amount, regardless of the outcome.
```

```
            lockedInBets = safeSub(lockedInBets, bet.lockedAmount);
            bet.lockedAmount= 0;
            // Send the funds to gambler.
            sendFunds(gambler,amount);
        }
        // Check that the bet has won by player.
        require (bet.winamount != 0, "winning amount can not be zero");
        // uint amoun = bet.amount;
        gambler = bet.gambler;
        uint winamount = bet.winamount;
        // Move bet into 'processed' state already.
        bet.amount = 0;
        bet.winamount = 0;
        // Unlock the bet amount, regardless of the outcome.
        lockedInBets = safeSub(lockedInBets, bet.lockedAmount);
        bet.lockedAmount= 0;

        // Send the funds to gambler.
        sendFunds(gambler, winamount);
    }

    // Helper routine to process the payment.

    function sendFunds(address beneficiary, uint amount) private{
        if (address(uint160(beneficiary)).send(amount)) {
            emit Payment(beneficiary, amount, true, 'successs');
        } else {
            emit Payment(beneficiary, 0, false, 'failure');
        }
    }

    /* only owner address can transfer ether */
    function ownerTransferEther(address sendTo, uint amount) public
        onlyOwner
    {
        require( sendTo != address(0), "address can't be zero");
        if(!address(uint160(sendTo)).send(amount)) revert();
        emit LogOwnerTransfer(sendTo, amount);
    }

    // Get the expected win amount after house edge is subtracted.

    function getDiceWinAmount(uint amount) private pure returns (uint winAmount) {
```

```
        uint houseEdge = amount * HOUSE_EDGE_PERCENT / 100;
        require (houseEdge <= amount, "Bet doesn't even cover house edge.");
        winAmount = safeSub(amount,houseEdge);
        return winAmount;
    }

}
```