Introduction to Blockchain and Smart Contracts – Final Project

Group 22 – Arda Harman

# Introduction

- DAO Governance Voting System

DAO Governance is a voting system. At this system, some proposals are proposed. These proposals are either get approved or not getting approved

- Proposals get approved by votes
    - If a proposal is not approved, it doesn't mean it got "rejected"
        - There is only "voting for a proposal" but there isn't "voting against a proposal"
        - If a person does not want a proposal to be approved, they just don't vote
            - They don't submit a vote against proposal, since there is nothing like that
    - It means that it didn't got enough votes "to getting approved"

- Participants

There are also some participants which participates in this system by voting. Participants cast their votes

- Tokens & Casting Votes

Tokens are used for casting votes. Participants obtain tokens with real money (like ether etc.) and they can use their obtained tokens for casting votes

Cost of a vote in terms of tokens grow quadratically for a proposal:

- 1 vote (for 1 proposal) -> 1 token
- 2 vote (for 1 proposal) -> 4 token
- 3 vote (for 1 proposal) -> 9 token
- 2 vote (1 vote for proposal A and 1 vote for proposal A) -> 1 + 1 = 2 token
- 3 vote (2 vote for proposal A and 1 vote for proposal A) -> 4 + 1 = 5 token

- "Total initial budget" & "budget"

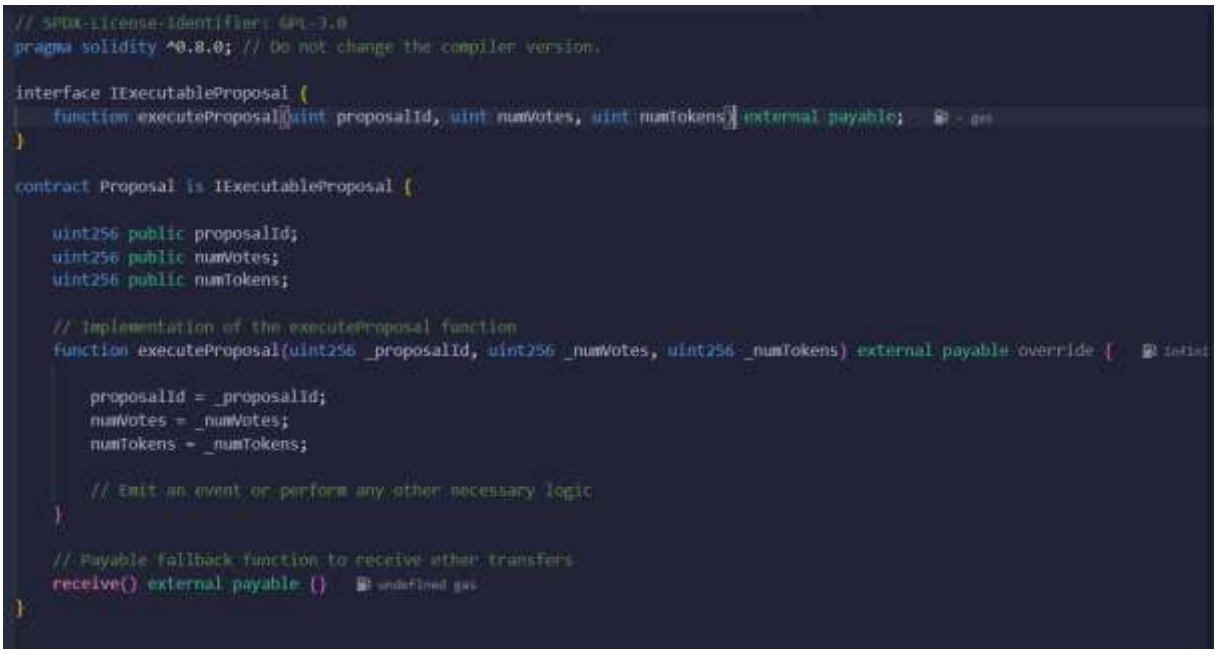
All proposals are proposed with a cost. That cost is called "budget". This budget is obtained from "total initial budget" meaning that the budget can't be more than "total initial budget".

- Proposal

There are 2 kinds of proposals:

  - “Funding” proposals
    - These proposals do have a budget and they immediatly get executed when they exceed threshoold
  - “Signaling” proposals
    - These proposals don’t have a specific budget but they are only there to express opinions. They aren’t executed until voting is closed

2.1 IExecutableProposal Interface & Proposal Contract



```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0; // Do not change the compiler version.

interface IExecutableProposal {
    function executeProposal(uint proposalId, uint numVotes, uint numTokens) external payable;
}

contract Proposal is IExecutableProposal {

    uint256 public proposalId;
    uint256 public numVotes;
    uint256 public numTokens;

    // Implementation of the executeProposal function
    function executeProposal(uint256 _proposalId, uint256 _numVotes, uint256 _numTokens) external payable override {

        proposalId = _proposalId;
        numVotes = _numVotes;
        numTokens = _numTokens;

        // Emit an event or perform any other necessary logic
    }

    // Payable fallback function to receive ether transfers
    receive() external payable {}
}
```

## 2.2 QuadraticVoting

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0; // Do not change the compiler version.

import "./VotingToken.sol"; // Import the ERC20 token contract
import "./IExecutableProposal.sol";

contract QuadraticVoting {

    address payable public owner;
    address[] public participants;

    bool public isVotingOpen;
    bool public isVotingClosed;

    uint256 public totalInitialBudget;
    uint256 public tokenPrice;

    uint public maxTokens;

    VotingToken public tokenContract;

    // Struct to represent a proposal
    struct Proposal {
        uint256 id;
        string title;
        string description;
        uint256 budget;
        address payee;
        address creator; // Address of the creator of the proposal
        bool isApproved;
        address[] voters;
        mapping(address => uint256) votes;
    }

    // Array to store all proposals
    Proposal[] public proposals;

    constructor(uint256 _pricePerToken, uint256 _maxTokens) {
        require(_pricePerToken > 0, "Prices of the token must be greater than 0");
        require(_maxTokens > 0, "Max tokens must be greater than 0");

        owner = payable(msg.sender);

        tokenPrice = _pricePerToken;
        // Create an instance of the ERC20 token contract
        tokenContract = new VotingToken("Voting Token", "VOTE");
        // Mint tokens for the contract creator

        maxTokens = _maxTokens;

        isVotingOpen = false;
        isVotingClosed = false;
    }

    function openVoting() external payable {
        require(msg.sender == owner, "Someone who is not a president is trying to OPEN the voting. Only president can OPEN voting");
        // Transfer the total initial budget for funding proposals
        uint256 _initialBudget = msg.value / 1 ether;

        require(_initialBudget > 0, "Initial budget must be greater than zero");
        require(address(this).balance >= _initialBudget, "Insufficient balance in contract");
        totalInitialBudget = _initialBudget;
        // Start the voting period
        // Additional logic can be added here
        isVotingOpen = true;
        isVotingClosed = false;
    }

    function addParticipant() external {
        require(!isParticipant(msg.sender), "Participant is already added!!");
        require(msg.sender != owner, "President can not be added as a participant");

        participants.push(msg.sender);
    }

    function isParticipant(address _address) internal view returns (bool) {
        for (uint256 i = 0; i < participants.length; i++) {
            if (participants[i] == _address) {
                return true;
            }
        }
        return false;
    }
```

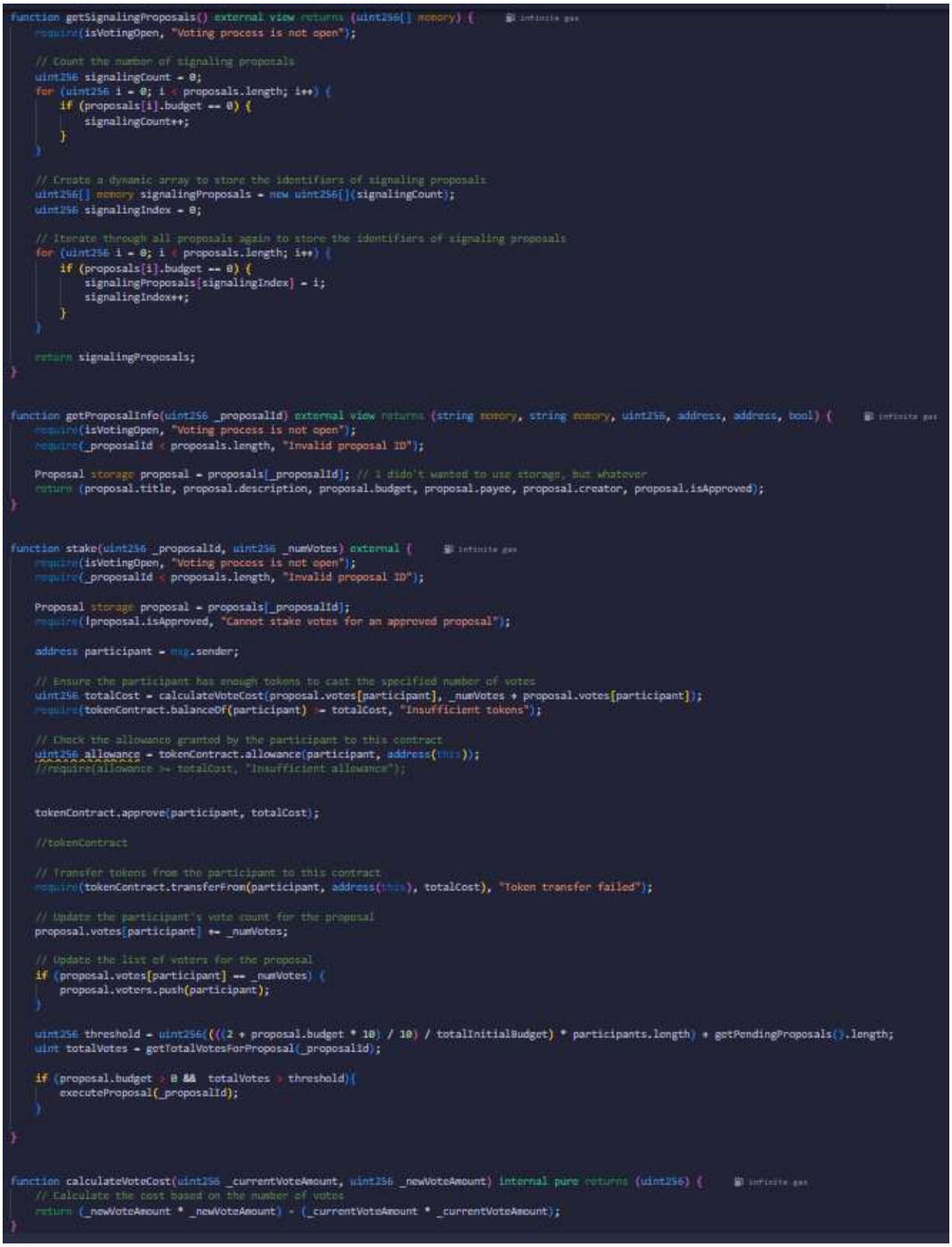
- Proposal: It represent the “proposals” and it has many attributes and each proposal is stored at proposals array
- openVoting: Starts the voting process and totalInitialBudget (only owner can execute it)
- totalInitialBudget: The total budget for proposing each proposal
- addParticipant: Adds participant
- isParticipant: Checks whether given person is already “added as a participant” or not,

```
function getTotalVotesForProposal(uint256 _proposalId) public view returns (uint256) {
    require(_proposalId < proposals.length, "Invalid proposal ID");

    Proposal storage proposal = proposals[_proposalId];
    uint256 totalVotes = 0;

    // Iterate through all voters for the proposal and sum their votes
    for (uint256 i = 0; i < proposal.voters.length; i++) {
        address voter = proposal.voters[i];
        totalVotes += proposal.votes[voter];
    }

    return totalVotes;
}

function removeParticipant() external {
    require(isParticipant(msg.sender), "Participant is not registered");

    // Burn all tokens owned by the participant
    uint256 tokenBalance = tokenContract.balanceOf(msg.sender);
    tokenContract.burn(tokenBalance);

    // Remove participant from the array
    for (uint256 i = 0; i < participants.length; i++) {
        if (participants[i] == msg.sender) {
            // Move the last element to the position of the removed element
            participants[i] = participants[participants.length - 1];
            // Remove the last element
            participants.pop();
            break;
        }
    }



// Function to add a proposal
function addProposal(string memory _title, string memory _description, uint256 _budget, address _payee) external returns (uint256) {
    require(isVotingOpen, "Voting process is not open");
    require(isParticipant(msg.sender), "Only registered participants can create proposals");
    require(totalInitialBudget >= _budget, "Budget for the proposal can not exceed total initial budget");

    // Increment proposal ID
    uint256 proposalId = proposals.length;

    // Create the proposal and add it to the array
    Proposal storage newProposal = proposals.push();
    newProposal.id = proposalId;
    newProposal.title = _title;
    newProposal.description = _description;
    newProposal.budget = _budget;
    newProposal.payee = _payee;
    newProposal.creator = msg.sender; // Set the creator's address
    newProposal.isApproved = false;

    return proposalId;
}


function cancelProposal(uint256 _proposalId) external {
    require(isVotingOpen, "Voting process is not open");
    require(_proposalId < proposals.length, "Invalid proposal ID");

    Proposal storage proposal = proposals[_proposalId];
    require(msg.sender == proposal.creator, "Only the proposal creator can cancel the proposal");
    require(!proposal.isApproved, "Approved proposals cannot be cancelled");

    // Iterate over voters and return their tokens
    for (uint256 i = 0; i < proposal.voters.length; i++) {
        address voter = proposal.voters[i];
        uint256 tokens = proposal.votes[voter];
        tokenContract.transfer(voter, tokens);
        delete proposal.votes[voter]; // Remove voter's vote from the votes mapping
    }

    // Remove the proposal
    delete proposals[_proposalId];
}
```

- getTotalVotesForProposal: Returns the total amount of the casted votes
- addProposal: Adds the proposal if it doesn’t have any flaws
- removeParticipant: Removes the given participant
- cancelProposal: It cancels the given proposal if it exists

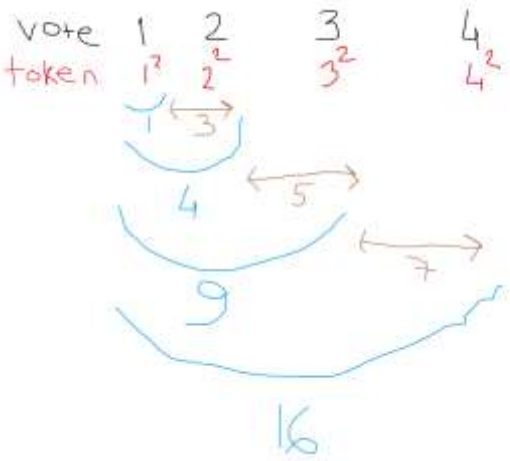
```
function buyTokens() external payable {
    require(isVotingOpen, "Voting process is not open");
    require(isParticipant(msg.sender), "A person who is not added as participant yet can not buy tokens");
    require(msg.value > 0, "Ether value must be greater than 0");

    // Calculate the number of tokens based on the token price
    uint256 tokenAmount = (msg.value / 1 ether) / tokenPrice; // Assuming tokenPrice is defined somewhere

    // Ensure that at least one token is bought
    require(tokenAmount > 0, "At least one token must be bought");

    require(maxTokens >= tokenAmount, "Tokens intended to be bought can not exceed maxTokens");

    // Transfer tokens to the buyer
    tokenContract.mint(msg.sender, tokenAmount);
}

function sellTokens(uint256 _tokenAmount) external {
    require(isVotingOpen, "Voting process is not open");
    require(_tokenAmount > 0, "Token amount must be greater than 0");

    // Transfer tokens from the participant to the contract
    tokenContract.transferFrom(msg.sender, address(this), _tokenAmount);

    // Calculate the amount of ether to send back to the participant
    uint256 etherAmount = _tokenAmount * tokenPrice * 1 ether; // Assuming tokenPrice is defined somewhere

    // Send ether back to the participant
    payable(msg.sender).transfer(etherAmount);
}

// gerekirse public yerine external yap
function getPendingProposals() public view returns (uint256[] memory) {
    require(isVotingOpen, "Voting process is not open");

    // Count the number of pending funding proposals
    uint256 pendingCount = 0;
    for (uint256 i = 0; i < proposals.length; i++) {
        if (!proposals[i].isApproved && proposals[i].budget > 0) {
            pendingCount++;
        }
    }

    // Create a dynamic array to store the identifiers of pending funding proposals
    uint256[] memory pendingProposals = new uint256[](pendingCount);
    uint256 pendingIndex = 0;

    // Iterate through all proposals again to store the identifiers of pending funding proposals
    for (uint256 i = 0; i < proposals.length; i++) {
        if (!proposals[i].isApproved && proposals[i].budget > 0) {
            pendingProposals[pendingIndex] = i;
            pendingIndex++;
        }
    }

    return pendingProposals;
}


function getApprovedProposals() external view returns (uint256[] memory) {
    require(isVotingOpen, "Voting process is not open");

    // Count the number of approved funding proposals
    uint256 approvedCount = 0;
    for (uint256 i = 0; i < proposals.length; i++) {
        if (proposals[i].isApproved) {
            approvedCount++;
        }
    }

    // Create a dynamic array to store the identifiers of approved funding proposals
    uint256[] memory approvedProposals = new uint256[](approvedCount);
    uint256 approvedIndex = 0;

    // Iterate through all proposals to store the identifiers of approved funding proposals
    for (uint256 i = 0; i < proposals.length; i++) {
        if (proposals[i].isApproved) {
            approvedProposals[approvedIndex] = i;
            approvedIndex++;
        }
    }

    return approvedProposals;
```

- buyTokens: It makes the participant buy a token with their money
- sellTokens: It makes a user sell their tokens
- getPendingProposals: Returns the id’s of all of the pending proposals
- getApprovedProposals: Returns the id’s of all of the approved proposals



```
function getSignalingProposals() external view returns (uint256[] memory) {
    require(isVotingOpen, "Voting process is not open");

    // Count the number of signaling proposals
    uint256 signalingCount = 0;
    for (uint256 i = 0; i < proposals.length; i++) {
        if (proposals[i].budget == 0) {
            signalingCount++;
        }
    }

    // Create a dynamic array to store the identifiers of signaling proposals
    uint256[] memory signalingProposals = new uint256[](signalingCount);
    uint256 signalingIndex = 0;

    // Iterate through all proposals again to store the identifiers of signaling proposals
    for (uint256 i = 0; i < proposals.length; i++) {
        if (proposals[i].budget == 0) {
            signalingProposals[signalingIndex] = i;
            signalingIndex++;
        }
    }

    return signalingProposals;
}

function getProposalInfo(uint256 _proposalId) external view returns (string memory, string memory, uint256, address, address, bool) {
    require(isVotingOpen, "Voting process is not open");
    require(_proposalId < proposals.length, "Invalid proposal ID");

    Proposal storage proposal = proposals[_proposalId]; // I didn't wanted to use storage, but whatever
    return (proposal.title, proposal.description, proposal.budget, proposal.payee, proposal.creator, proposal.isApproved);
}


function stake(uint256 _proposalId, uint256 _numVotes) external {
    require(isVotingOpen, "Voting process is not open");
    require(_proposalId < proposals.length, "Invalid proposal ID");

    Proposal storage proposal = proposals[_proposalId];
    require(!proposal.isApproved, "Cannot stake votes for an approved proposal");

    address participant = msg.sender;

    // Ensure the participant has enough tokens to cast the specified number of votes
    uint256 totalCost = calculateVoteCost(proposal.votes[participant], _numVotes + proposal.votes[participant]);
    require(tokenContract.balanceOf(participant) >= totalCost, "Insufficient tokens");

    // Check the allowance granted by the participant to this contract
    uint256 allowance = tokenContract.allowance(participant, address(this));
    //require(allowance >= totalCost, "Insufficient allowance");


    tokenContract.approve(participant, totalCost);

    //tokenContract

    // Transfer tokens from the participant to this contract
    require(tokenContract.transferFrom(participant, address(this), totalCost), "Token transfer failed");

    // Update the participant's vote count for the proposal
    proposal.votes[participant] += _numVotes;

    // Update the list of voters for the proposal
    if (proposal.votes[participant] == _numVotes) {
        proposal.voters.push(participant);
    }

    uint256 threshold = uint256(((2 + proposal.budget * 10) / 10) / totalInitialBudget) * participants.length) + getPendingProposals().length;
    uint totalVotes = getTotalVotesForProposal(_proposalId);

    if (proposal.budget > 0 &&  totalVotes > threshold){
        executeProposal(_proposalId);
    }

}


function calculateVoteCost(uint256 _currentVoteAmount, uint256 _newVoteAmount) internal pure returns (uint256) {
    // Calculate the cost based on the number of votes
    return (_newVoteAmount * _newVoteAmount) - (_currentVoteAmount * _currentVoteAmount);
}
```

- getSignalingProposals: Returnd the id's of all of the signaling proposals
- getProposalInfo: Returns all of the information about a proposal related about a proposal
- stake: This function allows a participant to cast a vote
- calculateVoteCost: It calculates the cost of a user to cast their vote
    - The resason it returns "$(\text{newVoteAmount})^2 - (\text{currentVoteAmount})^2$" is this:
    - With each vote, the cost increases quadratically
        - If a user voted totally 3 votes, in total he should spend totally 9 tokens
        - -> If a user first votes "1 vote" and then "2 votes" he should still spend 9 tokens
            - Not "$1^2 + 2^2 = 5$" tokens


vote 1 2 3 4
token 1² 2² 3² 4²
1
3
4
5
9
7
16

```
function getERC20() internal view returns (ERC20) {     infinite gas
    return ERC20(tokenContract);
}

function withdrawFromProposal(uint256 _proposalId, uint256 _numVotes) external {     infinite gas
    require(isVotingOpen, "Voting process is not open");
    require(_proposalId < proposals.length, "Invalid proposal ID");

    Proposal storage proposal = proposals[_proposalId];
    require(!proposal.isApproved && !isVotingClosed, "Cannot withdraw from an approved or closed proposal");

    address participant = msg.sender;

    // Ensure the participant has previously casted votes for this proposal
    require(proposal.votes[participant] >= _numVotes, "Insufficient votes to withdraw");

    // Calculate the number of tokens to return to the participant
    uint256 tokensToReturn = _numVotes * _numVotes;

    // Return the tokens to the participant
    require(tokenContract.transfer(participant, tokensToReturn), "Token transfer failed");

    // Reduce the participant's vote count for the proposal
    proposal.votes[participant] -= _numVotes;
}


function _checkAndExecuteProposal(uint256 _proposalId) internal {     infinite gas
    Proposal storage proposal = proposals[_proposalId];

    // Ensure the proposal is not already approved or cancelled
    require(!proposal.isApproved && !isVotingClosed, "Proposal already approved or voting process closed");

    // Check if the proposal is a signaling proposal
    if (proposal.budget == 0) {
        // Execute signaling proposal
        executeProposal(_proposalId);
        return;
    }

    // ((2 + proposal.budget * 10) / 10)
    // Compute the threshold for the proposal
    uint256 threshold = uint256(((2 + proposal.budget * 10) / 10) / totalInitialBudget) * participants.length) + getPendingProposals().length;

    // Check if the proposal received enough votes to exceed the threshold
    if (proposal.voters.length > threshold) {
        // Execute funding proposal
        executeProposal(_proposalId);
    }
}

bool lock = false;

// Function to execute a proposal
function executeProposal(uint256 _proposalId) internal {     infinite gas

    require(!lock, "Contract locked!!");

    Proposal storage proposal = proposals[_proposalId];

    // Call the executeProposal function of the external contract through the interface

    IExecutableProposal externalContract = IExecutableProposal(proposal.payee);
    lock = true;
    externalContract.executeProposal{value: proposal.budget * 1 ether}(_proposalId, getTotalVotesForProposal(_proposalId), getTotalVotesForProposal(_proposalId)+getTotalVotesForProposal(_proposalId));
    lock = false;

    // Update total budget and remove tokens related to the proposal
    totalInitialBudget -= proposal.budget;
    totalInitialBudget += calculateTotalSpentTokensForProposal(_proposalId) * tokenPrice;

    for (uint256 i = 0; i < proposal.voters.length; i++) {
        address voter = proposal.voters[i];
        uint256 tokens = proposal.votes[voter];
        tokenContract.transfer(voter, tokens);
        delete proposal.votes[voter];
    }

    // Mark the proposal as approved
    proposal.isApproved = true;
}
```

- getERC20: Returns the used tokenContract
- withdrawFromProposal: It allows a user to wihtdraw from a propsala and also withdraws his votes and returns his tokens back to them
- executeProposal: It executes a given proposal
  - If executed proposal is a funding proposal, then it also turns that into “approved”
  - If executed proposal is a signaling proposal, then it executes it (after closeVoting is executed) and these proposals got deleted
  - Also, after execution:
    - The budget is transfered from “QuadraticVoting” contract to the “Proposal” contract
    - totalInitialBudget gets recalculated

```
    function closeVoting() external payable {      infinite gas
        require(msg.sender == owner, "Only the owner can close the voting period");
        require(isVotingOpen, "Voting period is not open");

        // Dismiss funding proposals that have not been approved
        for (uint256 i = 0; i < proposals.length; i++) {
            Proposal storage proposal = proposals[i];
            if (proposal.budget > 0 && !proposal.isApproved) {
                for (uint256 j = 0; j < proposal.voters.length; j++) {
                    address voter = proposal.voters[j];
                    uint256 tokens = proposal.votes[voter];
                    tokenContract.transfer(voter, tokens);
                    delete proposal.votes[voter];
                }
                delete proposals[i];
            }
        }

        // Execute signaling proposals and return tokens
        for (uint256 i = 0; i < proposals.length; i++) {
            Proposal storage proposal = proposals[i];
            if (proposal.budget == 0) {
                executeProposal(i);
            }
        }

        // Transfer remaining voting budget to owner
        owner.transfer(totalInitialBudget * 1 ether);

        // Reset contract state
        isVotingOpen = false;
        isVotingClosed = true;

        totalInitialBudget = 0;

        delete proposals;
        delete participants;
    }

    function calculateTotalSpentTokensForProposal(uint256 _proposalId) internal view returns (uint){
        Proposal storage proposal = proposals[_proposalId];

        uint tokenAmount = 0;

        for (uint256 j = 0; j < proposal.voters.length; j++) {
            address voter = proposal.voters[j];

            tokenAmount += proposal.votes[voter] * proposal.votes[voter];
        }

        return tokenAmount;
    }

}
```

- closeVoting: Voting process gets closed and the following operations happen:
  - Signaling proposals get executed
  - Funding proposals which are not approved gets deleted and users get their tokens back
  - The totalInitialBudget is transfered to the owner of the QuadraticVotingToken

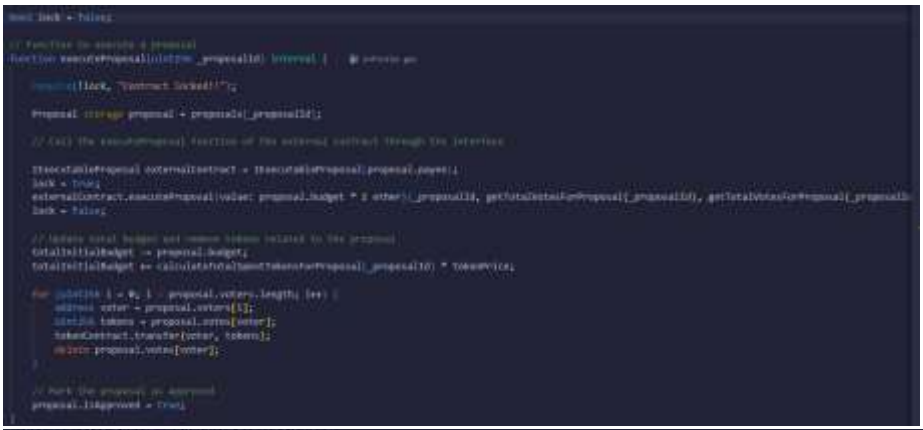
3. ERC20 Token & VotingToken contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract VotingToken is ERC20 {
    address public owner;
    address public contractAddress;
    uint public a = 2;

    constructor(string memory name, string memory symbol) ERC20(name, symbol) {
        owner = msg.sender; // Set contract deployer as owner
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner can perform this action");
        _;
    }

    function copyVotingToken(address _contractAddress) external onlyOwner returns (VotingToken){
        require(contractAddress == _contractAddress, "Contract ");
    }

    // External function to mint new tokens for participants
    function mint(address account, uint256 amount) external onlyOwner {
        _mint(account, amount);
    }

    // Control who can burn tokens
    function burn(uint256 amount) external {
        _burn(msg.sender, amount);
    }

/*
    // Function to approve spending tokens by another contract
    function approve(address owneerr, address spender, uint256 amount) public returns (bool) {
        _approve(owneerr, spender, amount);
        return true;
    }
*/

    function approve(address owneerr, uint256 amount) public override returns (bool) {
        _approve(owneerr, msg.sender, amount);
        return true;
    }
}
```

- Constructor: It takes “msg.sender” because the constructor gets executed by QuadraticVoting contract
    - So it will be assigned with the address of the QuadraticVoting contract
    - This makes that “VotingToken”s owner will be QuadraticVoting
        - This makes sense, since participants operate with tokens AND THESE TOKENS ARE PROVIDED BY QuadraticVoting contract
            - Also, QuadraticVoting contract should allow users to operate with tokens
- burn: It burns the tokens of the given contract
    - It had to be written again, because otherwise there is no way to call burn function from ERC20 contract
- mint: It transfers token to the given address
    - It had to be written again, because otherwise there is no way to call mint function from ERC20 contract
- approve: It gives allowance to the given spender

4. Security Measures

-> Reentracy Attack



```
contract Proposal is IExecutableProposal {

    uint256 public proposalId;
    uint256 public numVotes;
    uint256 public numTokens;

    // Implementation of the executeProposal function
    function executeProposal(uint256 _proposalId, uint256 _numVotes, uint256 _numTokens) external payable override {

        proposalId = _proposalId;
        numVotes = _numVotes;
        numTokens = _numTokens;

        // Emit an event or perform any other necessary logic
    }

    // Payable fallback function to receive ether transfers
    receive() external payable {}
}
```

Recieve function doesn't call executeProposal, so it is not directly vulnerable to "reentracy". However, since at the project, it was requested to fixing some security issues, I felt the need to add "lock-based mechanism" for reentracy attack

-> ParityWallet attack:

```
contract VotingToken is ERC20 {
    address public owner;
    address public contractAddress;
    uint public a = 2;

    constructor(string memory name, string memory symbol) ERC20(name, symbol) {
        owner = msg.sender; // Set contract deployer as owner
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner can perform this action");
        _;
    }
```

Again, this code isn't directly vulnerable to ParityWallet attack, but with the modifier, it is even stronger against it

-> Overflow/underflow attack: I already use version 8.0.20 but I could have also implemented "SafeMath" but I didn't since it is not needed