Google Summer of Code

GSOC'22 PROPOSAL - AOSSIE

Agora Blockchain Project

Organization: AOSSIE

Project Name: Agora Blockchain

Candidate Name: Sreeniketh Madgula

AOSSIE Project Description: Here

Agora Blockchain Issue: #11

Mentors: Raj Ranjan

Expected Project Size: 175 hours

Contents

1. About me

Name: Sreeniketh Madgula

Email: sreeniketh.madgula@gmail.com

2. How many hours I can work per week

I will work for an average of 28-30 hours every week.

3. Other commitments

1. College Exams
2. Course projects

These are going to be an integral part of my 6th semester activities. They are not going to affect my GSOC efforts.
I will be able to deliver my commited number of hours per week without much hindrance.

4. Chosen Idea

I wish to contribute to Agora Blockchain project.

Abstract

- Refine smart contracts to make codebase extensible to new features
- Follow SOLID principles during the design of smart contracts
- Implement different voting algorithms like **Moore**, **Oklahoma** etc.
- Integrate authentication and user KYC
- Extend support for open and invite-based elections

5. Proposal Description

Summary

Agora Blockchain is a decentralized platform for conducting elections. Agora is a library of voting algorithms like Moore's, Oklahoma etc. Some of these voting algorithms are already implemented by AOSSIE in a centralized manner using Scala as their backend. The current version of the DApp has only one voting algorithm - **General**. The idea is to extend support to other electoral systems - Majoritarian, Preferential(**Oklahoma**), Plurality etc. and result computation algorithms like **Moore**.

5.2. Refining Smart Contracts - for multiple algorithm integration

> - Currently, the Election.sol smart contract accounts only for general election where a user can vote for 1 person

> - The idea is to have multiple voting algorithms like **Moore's**, **Oklahoma** etc.
> - The possible implementation was to have algorithm-specific smart contracts, i.e. separate smart contracts for different algorithms

Smart contracts are analogous to classes in object oriented languages. They contain state variables that can manipulate data and functions that can manipulate data in state variables.
Extensibility is the key to building large, complex dapps. Interfaces help achieve abstraction and extensibility.

5.2.1. Interfaces

They act as blueprints that are used to specify the behaviour of contracts.
An interface is a mechanism to achieve abstraction. In other words an interface is used to decouple the behaviour of a contract from its implementation. So, an interface cannot have any of its functions implemented. As a result, it cannot be compiled.
There are a few restrictions of an interface:

- They cannot inherit from other contracts, but they can inherit from other interfaces.
- All declared functions must be external in the interface, even if they are public in the contract.
- They cannot declare a constructor and hence cannot be instantiated.
- They cannot declare state variables.
- They cannot declare modifiers.

Interfaces are basically limited to what the Contract **ABI** can represent, and the conversion between the ABI and an interface should be possible without any information loss. Interfaces are denoted by their own keyword. Here is an example interface:

```
pragma solidity >=0.6.2 <0.9.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

- Clearly, no state variable has been declared. The **enum** and **struct** can not have a variable as part of the interface. They must be instantiated only in the subcontracts. Similarly, the function implementation must also be defined only in subclasses.
- All functions declared are considered **virtual** implicitly, and functions that override them do not need the **override** keyword. However, to override an overriding function again, it must be declared **virtual**.

The architecture of the smart contracts should be according to SOLID principles.

5.2.2. SOLID Principles

The smart contracts are akin to classes in object oriented programming languages.

- **S** - Single Responsibility Principle
    - This states that a class(contract) should have only one job, i.e., one main function

- **O** - Open Close Principle
    - This states that a contract should be open for extension but closed for modification
    - Currently, the **Election.sol** contract violates this principle. It is not extendable to implement multiple voting algorithms.
- **L** - Liskov Substitution Principle
    - Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.
    - This means that every subcontract or derived contract should be substitutable for their base or parent contract.
- **I** - Interface Segregation Principle
    - A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use.
- **D** - Dependency Inversion Principle
    - Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions
    - This principle allows for decoupling.

More about SOLID principles

5.2.3. Current Design Issues

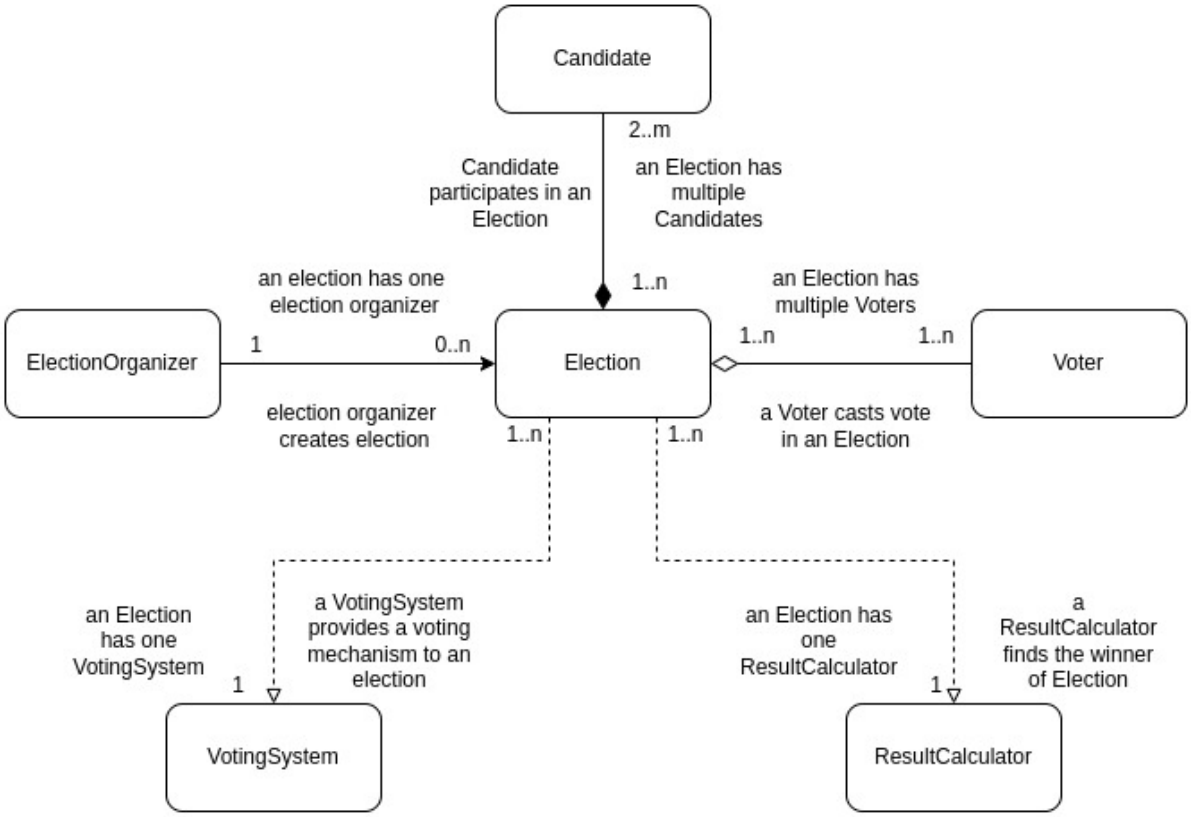
Currently the architecture is such that each of smart contracts performs the essential functions that fulfill the basic features of the application.
These are the issues with the present architecture:

- Although the current design of the smart contracts satisfies the existing use case, it is not extensible, i.e., it is not easy to add new features.
- Smart contracts require to be modified each time a new feature has to be supported.
- This could mean more frequent breaking changes to the application, which makes old data incompatible.
- Realizing the **Open-Close** principle, it should be possible to integrate a new feature without having to make changes to the current application.
- Semantic Versioning standards can be adopted to the smart contracts.

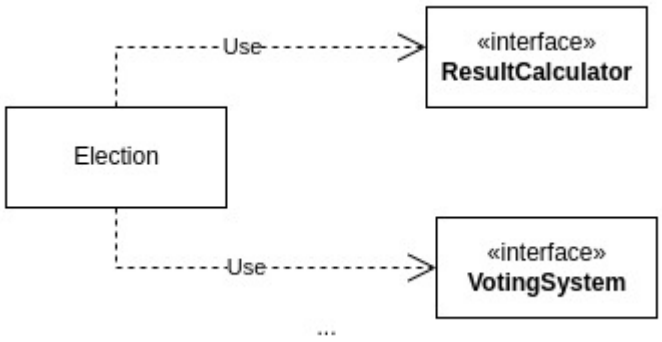
The deployment of code costs gas linear to the length of the code. To implement a new feature, with the present architecture would have a lot of redundancy and code duplication.

5.3. Object Model

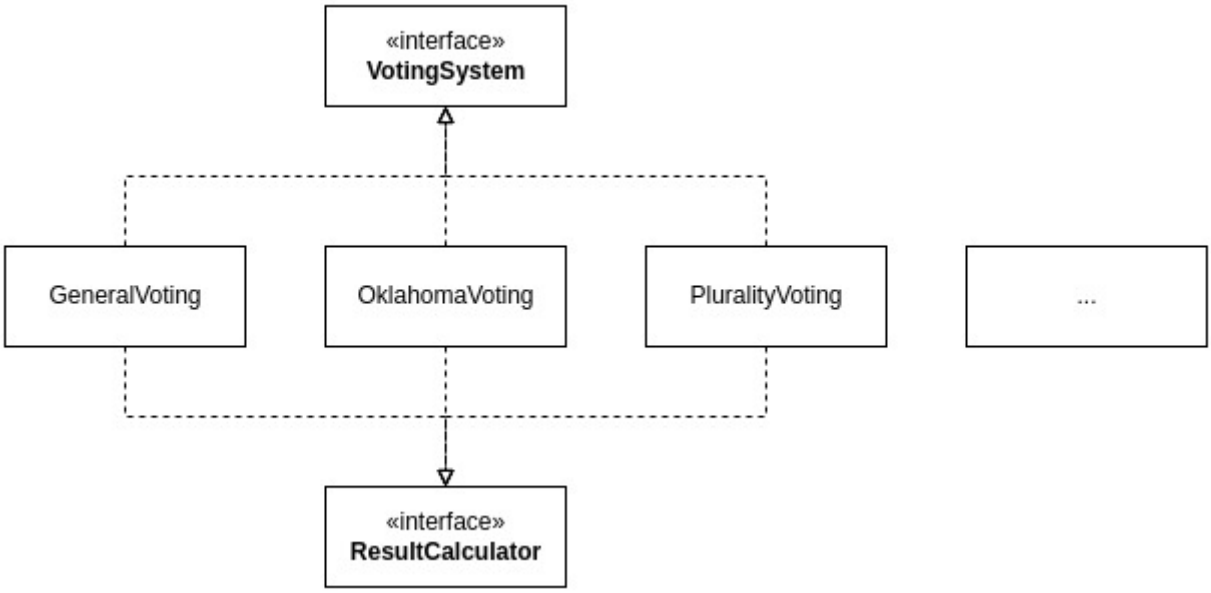
To refine the smart contracts, the object model of the modified architecture would look like:



The **Election** contract has dependencies on **VotingSystem** and **ResultCalculator** interfaces:



VotingSystem interface has the method **vote()** and **ResultCalculator** interface has **getResult()** method. These methods are implemented by each algorithm-specific contracts:



Based on the object model, the smart contracts would be:

- **ElectionOrganizer**
- **Candidate**
- **Voter**
- **Election**

The interfaces would be:

- **VotingSystem** with a **vote()** method
- **ResultCalculator** with a **getResults()** method

And the algorithm-specific smart contracts that implement **VotingSystem** and **ResultCalculator** would be:

- **GeneralElection**
- **OklahomaElection** - preferential voting; this would its own specific implementation of **getResult()**
- **MajoritarianElection** - where the **Moore** algorithm would be required in **getResult()**
- and so on

Additionally:

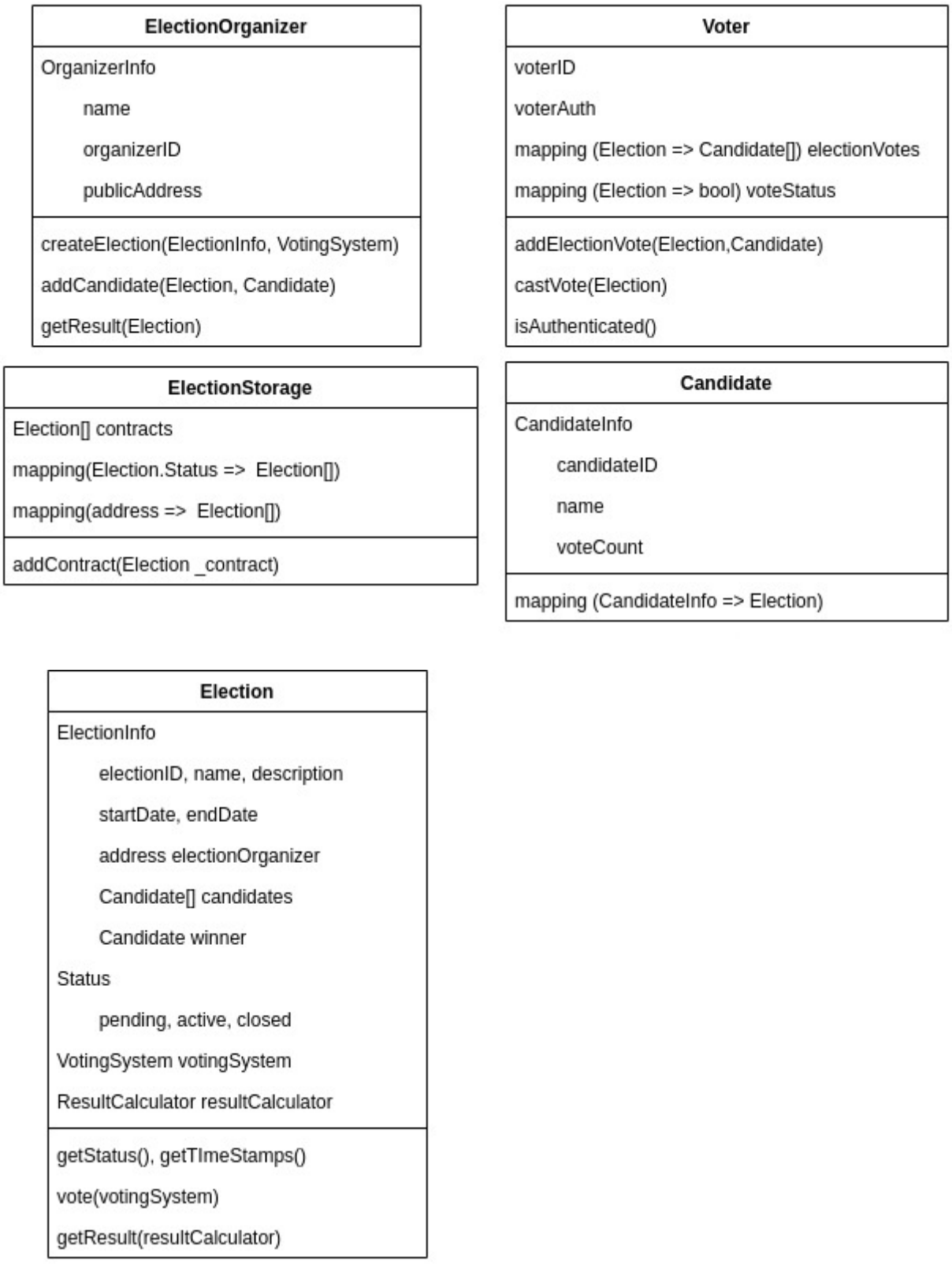
- **ElectionStorage** contract stores instances of the created elections with getters and setters.
- The **createElection()** method of **ElectionOrganizer** contract would add the **Election** instance to the state of **ElectionStorage**
- This allows access to each **Election** instance. The instances can further be segregated as **pending**, **active** and **closed** according to the **Status** enum.

  ```
  enum Status {active, pending, closed}
  ```

- The contract instances should be stored as two types of mappings

```
// Status to Election instance array
mapping (Election.Status => Election[]) contractsByStatus;
// election organizer address to Election instance array
mapping (address => Election[]) contractByOrganizer
```

Following the SOLID principles, these would be the attributes and behaviours of each contract:
Every contract has a constructor to initialize the state.



5.3.1. Voting System

There are different voting systems and different ways in which results of the election are declared

- Voting Systems:
    - **Plurality Systems**: Plurality voting is a system in which the candidate(s) with the highest number of votes wins, with no requirement to get a majority of votes
    - **Majoritarian Systems**: Majoritarian voting is a system in which candidates must receive a majority of votes to be elected, either in a runoff election or final round of voting (although in some cases only a plurality is required in the last round of voting if no candidate can achieve a majority).
    - **Ranked Voting**(**Oklahoma**): It is a type of Majoritarian voting in which voters rank their candidates (or option) in a sequence of 1st, 2nd, 3rd, etc.
    - and so on
- Result Calculators: Different voting systems have different ways of computing results.
- Currently there is only one type of algorithm - General
- To implement multiple algorithms, with the current design, **Election** smart contract has to be modified each time.
- Say we have made a change to **Election** contract, to include **Oklahoma** voting algorithm. This would require making changes to **vote()** function, and also to the getters and setters. Additionally, it would be a hassle to implement this in a backward compatible way.
- **Election** should have a **vote** functionality. It is not the responsibility of the **Election** object to be aware of the implementation details of the voting algorithm.
- Here, the voting algorithm is a low-level module and **Election** is high-level. But according to the **Dependency Inversion** principle, **Election** should depend on an abstraction rather than the implementation.
- To overcome this, a **VotingSystem** interface should be created that has a **vote()** method.
- Each of the specific algorithms would be separate contracts that have a **vote()** method. They would implement the **VotingSystem** interface.
- This makes the codebase open for extension (to support new algorithms) and closed for modification(**Open-Closed** principle)

vote() method:

- Each algorithm-specific contract implements the **vote()** method
- The signature of this method would be:

```
function vote(Election,Candidate[])
```

- A list of candidates is passed to support even **preferential voting**(**Oklahoma**) where the preferrences are recorded according to the index in the list.

5.3.2. Result Calculator

- Similarly, the **Election** object should not depend on the implementation of **getResult()** method. Different voting algorithms have different ways of computing the result.
- A **ResultCalculator** interface should be created with a **getResult()** method.
- The algorithm-specific smart contracts implement this interface and define the **getResult()** method accordingly.

This makes it easier to add new voting systems and result computation algorithms.
For each new pair, a contract would need to be created that implements these interfaces.

The **Election** contract would have a **VotingSystem** dependency and a **ResultCalculator** dependency.

5.3.3. Boyer-Moore Voting Algorithm:

- The Boyer-Moore voting algorithm is one of the popular optimal algorithms which is used to find the majority element among the given elements that have more than N/ 2 occurrences. This works perfectly fine for finding the majority element which takes 2 traversals over the given elements, which works in O(N) time complexity and O(1) space complexity.
- This algorithm works on the fact that if an element occurs more than N/2 times, it means that the remaining elements other than this would definitely be less than N/2. So let us check the proceeding of the algorithm.
- First, choose a candidate from the given set of elements if it is the same as the candidate element, increase the votes. Otherwise, decrease the votes if votes become 0, select another new element as the new candidate.

With a **ResultCalculator** interface, the **Moore** voting algorithm can easily be implemented.

- Create a **Moore** contract that implements **ResultCalculator** interface.
- The contract has a **getResult()** method that contains the **Boyer-Moore** result calculation algorithm to determine the election winner.

5.3.4. Access specifiers (modifiers)

Function Modifiers in Solidity
Modifiers can be used to change the behaviour of functions in a declarative way. For example, you can use a modifier to automatically check a condition prior to executing the function.

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

Function modifiers can be used to provide authorization. For example, an **registeredOnly** modifier can be used to restrict voters without completed KYC authentications to cast their votes.
organizerOnly modifier can restrict creation of elections only to users registered as election organizers

5.4. Workflow

1. Create election organizer
    - A user, once logged in would have options:
        - vote
        - create elction
    - Choosing create election should initialize the **ElectionOrganizer** contract and store details in a **struct**.
2. Organizer creates election
    - In **Create Election** the **organizer** would require to enter election details
        - name of the election
        - description of the election
        - start date and end date

        ```
        struct ElectionInfo {
            uint electionID;
            string name;
            string description;
            string algorithm;
            uint sdate;
            uint edate;
            uint voterCount;
            address electionOrganiser;
            Candidate[] candiates // this is an addition
            Candidate winner; // this is an addition
        }
        ```

    - The details would be used to instantiate the **Election** contract, and hence the **ElectionInfo** struct.
    - The organizer will be provided a list of **electoral systems** to choose from:
        - General
        - Oklahoma
        - Majoritarian
        - Plurality and so on
    - When the organizer chooses one of these, say **Oklahoma**, the application should create an instance of **OklahomaVoting** contract, say **oklahomaVoting**. This instance also has the implementation of **getResult()** method

- When all the details have been filled, the **createElection()** method of **ElectionOrganizer** contract is called and the details are passed to it along with the instance of the voting algorithm which contains **vote()** and **getResult()** methods, here **oklahomaVoting**.

3. Organizer adds candidates to the election
   - **Election** instances are accessible from **ElectionStorage** contract. These are displayed to the organizer as a list.
   - When the organizer chooses an election, the specific **Election** instance is acquired. This can then be passed to the **addCandidate()** method of **ElectionOrganizer** contract.
   - This can only be done before the start date of the election.
4. Voters cast votes
   - A list of active elections is displayed to the voter based on the start date of the elections.
   - This is accessible from the **contractsByStatus** mapping of **ElectionStorage** contract.
   - Once the voter selects the election to vote in, the instance of that specific election is stored as, say **currentElection** on the **web3** interface.
   - Then, the list of candidates in **currentElection** are displayed. According to the algorithm of that election.
   - Then according to the algorithm, the voter can choose candidates

     - In **General**, the voter selects the candidate to vote for.
     - The **addElectionVote()** method of **Voter** contract is called. This updates the **electionVotes** mapping to add the candidate to the **Candidate** list.

       ```
       mapping(Election => Candidate[]) electionVotes;
       ```

     - Then the **castVote()** method of **Voter** contract is called and passed with **currentElection**.
     - **castVote()**:

       ```
       // in Voter contract
       function castVote(Election _election) public {
         _elecion.vote(_election, electionVotes[_election]);
       }
       ```

     - The **vote** method of that specific election is called.

       ```
       // in Election contract
       function vote(Election _election, Candidate[] candidates)
      public {
         votingSystem.vote(_election,Candidate[]);
       }
       ```

     - **votingSystem** is the dependency in **Election** contract, of **VotingSystem** interface.

- The authentication of voters should be verified before they cast votes. No authentication required in invite-based elections.
- Voter is able to cast vote according to the **voting algorithm** implemented in the specific election

5. Election result is computed.

    - The **Election** instances are accessed from the **ElectionStorage** contract.
    - Organizer selects an election and the instance is stored as **currentElection**.
    - Once the end date of the election has passed, the result can be computed by the **getResult()** method.
    - Election organizer calls **getResult()** method of **ElectionOrganizer** contract:

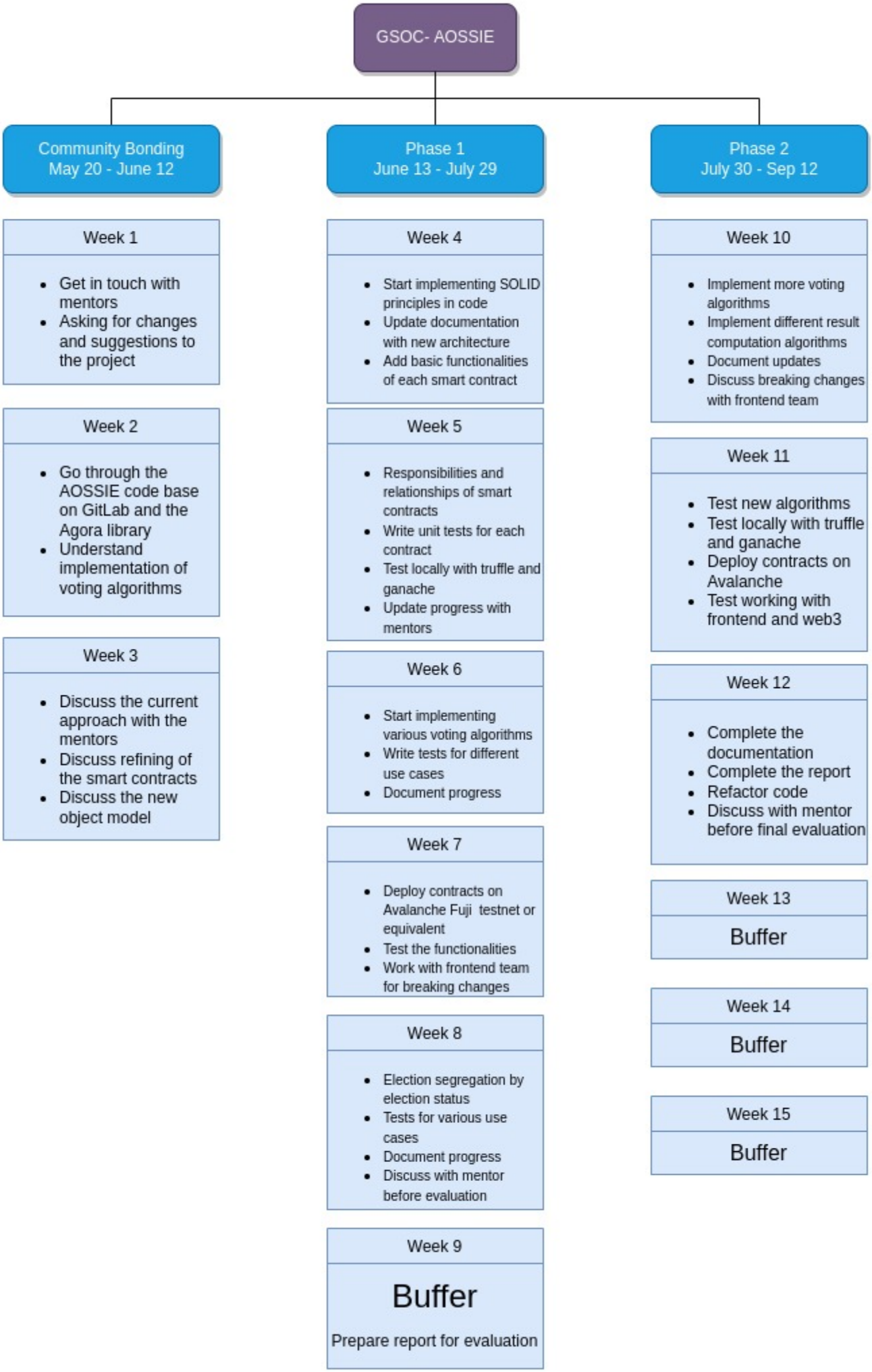
    ```
    // in ElectionOrganizer contract
    function getResult(Election _election) public {
      _election.getResult(_election);
    }
    ```

    - The **getResult** method of that specific election is called.

    ```
    // in Election contract
    function getResult(Election _election) public {
      resultCalculator.getResult(_election);
    }
    ```

    - resultCalculator is a dependency in **Election** contract, of **ResultCalculator** interface.
    - The **getResult()** method returns a **Candidate** instance, the winner of the election. The **winner** in the **ElectionInfo** struct is updated.

6. Week-wise Breakdown


GSOC- AOSSIE
Community Bonding
May 20 - June 12
Phase 1
June 13 - July 29
Phase 2
July 30 - Sep 12
Week 1
Get in touch with mentors
Asking for changes and suggestions to the project
Week 2
Go through the AOSSIE code base on GitLab and the Agora library
Understand implementation of voting algorithms
Week 3
Discuss the current approach with the mentors
Discuss refining of the smart contracts
Discuss the new object model
Week 4
Start implementing SOLID principles in code
Update documentation with new architecture
Add basic functionalities of each smart contract
Week 5
Responsibilities and relationships of smart contracts
Write unit tests for each contract
Test locally with truffle and ganache
Update progress with mentors
Week 6
Start implementing various voting algorithms
Write tests for different use cases
Document progress
Week 7
Deploy contracts on Avalanche Fuji testnet or equivalent
Test the functionalities
Work with frontend team for breaking changes
Week 8
Election segregation by election status
Tests for various use cases
Document progress
Discuss with mentor before evaluation
Week 9
Buffer
Prepare report for evaluation
Week 10
Implement more voting algorithms
Implement different result computation algorithms
Document updates
Discuss breaking changes with frontend team
Week 11
Test new algorithms
Test locally with truffle and ganache
Deploy contracts on Avalanche
Test working with frontend and web3
Week 12
Complete the documentation
Complete the report
Refactor code
Discuss with mentor before final evaluation
Week 13
Buffer
Week 14
Buffer
Week 15
Buffer

6.1. Community Bonding Period (May 20 - June 12)

Week 1

- Get in touch with the developers and the mentor
- Introduction to the community, to the mentor and fix timings to communicate
- Discuss any suggestions and changes to the project. There could modifications, new additions or ammendments; it would be better to go over these early

Week 2

- Go through the AOSSIE codebase and the Agora voting library
- Understand the implementation of various voting algorithms in the centralized backend application
- Complete any remaining local setup

Week 3

- Discuss the working of the existing application with the mentor. Discuss the implementations of the new features
- Discuss the refining of smart contracts and how new featue support is extended with the proposed architecture
- Talk over the new object model of the application and the use cases
- Go over the new design in detail and ask for changes and suggestions
- Discuss the changes that need to take place in the frontend to accomodate for the breaking changes

6.2. Phase 1 (June 13 - July 29)

Week 4

- Start laying out the smart contracts layer-wise. Lay out the foundation, with the basic features of each smart contract
- Follow SOLID principles while coding the smart contracts
- Start documenting the changes in architecture, the object model and about the new smart contracts

Week 5

- Work on the responsibilites and relationships and smart contracts to perform their primary functions
- Write unit tests for each of the smart contracts
- Test the working of the smart contracts with basic features locally with **truffle** and **ganache** wallet
- Update mentors with progress and discuss suggestions

Week 6

- Begin working on implementing various voting algorithms like **Moore**, **Oklahoma** etc.
- Write tests to test the working of these algorithms
- Test the **ResultCalculator** for different algorithms
- Document the new features being added and the tests written
- Work on migrations

Week 7

- Deploy the smart contracts on an RPC node (Avalanche Fuji testnet) with the first set of algorithms
- Test the working of the smart contract through the contract addresses deployed on the blockchain network
- Work closely with the frontend team and discuss the breaking changes in the application
- Share documentation and discuss testing the working with **web3**

Week 8

- Work on segregating elections based on election status - active, pending, closed based on start and end date
- Write tests for different use cases, creating different types of elections, testing segregation of elections etc
- Test locally with **truffle** and **ganache**
- Update the documentation with all new additions and modifications and discuss with frontend team
- Discuss progress with the mentors before the **Phase 1** evaluation

Week 9 and rest of Phase 1

- *Buffer* for any pending tasks
- Prepare a report for evaluation
- Discuss brief plan for Phase 2

6.3. Phase 2 (July 25 - Septempber 12)

Week 10

- Implement more algorithms of various electoral systems in the smart contracts

- Implement the result computation algorithms
- Work on use cases where multiple result computation algorithms are applicable to an election instance
- Document progress
- Update any breaking changes with frontend team for compatibility

Week 11

- Test the new voting algorithms and result computation algorithms
- Write tests for the different use cases.
- Test these locally with **truffle** and **ganache**
- Deploy the contracts on Avalanche, acquire tokens from the faucet, and test the deployed contracts
- Test the working of the deployed contracts and various use cases on the frontend with **web3**.

Week 12

- Complete the documentation to include the new algorithms and all the tests
- Refactor the code and complete the report
- Discuss with the mentor before final evaluation

Week 13 and rest of Phase 2

- *Buffer* to complete any remaining tasks

7. Relevant Skills and Experience

My team won a national level hackathon recently for building a decentralized taxi hailing service application named dRyver
You can find our project submission here

7.1. Skills

- Solidity
- web3
- truffle

7.2 Experience

- During my internship at Brane Enterprises, I worked on Java Spring and Spring boot to create RESTful APIs. I got to learn a lot about architectures of applications at enterprise-level.
- I learned about design principles and some advanced object oriented concepts.
- I am currently the **Secretary** of the **ACM Student Chapter** at my college and a memeber of **Google Developer Student Club**.

8. Projects

- dRyver
    - This is a decentralized taxi hailing service application
    - It is implemented using Ethereum smart contracts written in **Solidity**
    - The frontend is in **React.js**
    - Link to the repository

9. Contact

Name: Sreeniketh Madgula

Email: sreeniketh.madgula@gmail.com

LinkedIn: https://www.linkedin.com/in/sreenikethmadgula/

GitHub: https://github.com/sreenikethMadgula

Mobile Number: +91 9986020902