

**Facultad de Ingeniería**  
**Universidad ORT Uruguay**

**Tecnologías Blockchain para**  
**Contratos Inteligentes**

**Obligatorio**

Autores:

Alfonso Irazabal Levy - 201864

Francisco Esteban Cabanillas - 231918

Matias Wilner Merla - 231073

Profesor:

David Gimenez

2021

# Índice

<b>Índice</b>	<b>1</b>
<b>Descripción del trabajo</b>	<b>1</b>
<b>Justificaciones de diseño y cumplimiento de requerimientos</b>	<b>2</b>
<b>Arquitectura del proyecto</b>	<b>8</b>
Manejo de estados del sistema	9
<b>Testing</b>	<b>12</b>
<b>Deploy</b>	<b>14</b>
<b>Diagrama de Contratos</b>	<b>16</b>
Contrato SmartInvestment	16
Administración de roles	16
Métodos de la lógica	17
Contrato Proposal	17
Manejo de la propiedad de la propuesta	17
Lógica de votación	17
<b>Diagramas de Secuencia</b>	<b>18</b>
<b>Posibles mejoras y conclusiones</b>	<b>21</b>

# Descripción del trabajo

Se implementó un sistema de votación basado en la tecnología Blockchain. El sistema permite gestionar de forma descentralizada y autónoma la realización de votaciones a diferentes propuestas de inversión que son creadas con un nombre, descripción, inversión mínima requerida y un dueño.

El sistema es capaz de manejar distintos roles (owner, auditor, maker, voter), cada uno con sus respectivos permisos y permite el manejo de transferencias.

Fue desarrollado en el lenguaje de programación Solidity sobre la blockchain de Ethereum mediante contratos inteligentes. Además de eso, se desarrollaron también clases de pruebas unitarias en el lenguaje Javascript para tres métodos centrales del sistema:

- Creación de una propuesta
- Votar
- Cambiar de estado del sistema al estado de presentación de propuestas

Por otro lado, se desarrolló un script de deploy del proyecto para ser desplegado en la testnet de Ethereum (Ropsten). Las consideraciones del deploy serán desarrolladas en la sección "Deploy".

# Justificaciones de diseño y cumplimiento de requerimientos

En el proyecto se cumplieron todos los requerimientos de las tablas de Roles, Propuestas y Proceso de Votación expresadas en la letra del obligatorio.

Se adjuntan imágenes a continuación de los mismos y las resoluciones a nivel de diseño de la aplicación de los más relevantes:

- Roles:

ID	DESCRIPCIÓN
1	<b>ROL OWNER:</b> Son las cuentas de los dueños del sistema
2	<b>ROL MAKER:</b> Son las cuentas de las personas que crean las propuestas
3	<b>ROL AUDITOR:</b> Valida las propuestas presentadas
4	<b>ROL VOTER:</b> Cualquier cuenta que no pertenece a los Owners, Makers ni Auditors Son los únicos que pueden participar del proceso de votación.
5	Solo los owners pueden registrar a otros owners. Siendo el primer owner la cuenta del creador del contrato SmartInvestment
6	Solo los owners podrán registrar makers.
7	Solo los makers deben poder presentar propuestas
8	Deben registrarse al menos 3 makers para que el sistema habilite los procesos de propuesta y votación
9	Deben registrarse al menos 2 auditors para habilitar los procesos de propuesta y votación
10	Los datos de los makers que deben ser registrados son: <ul style="list-style-type: none"><li>• Nombre</li><li>• País de residencia</li><li>• Número de pasaporte</li></ul>

- Se utilizaron mappings con booleanos en lugar de enteros, con el objetivo de no utilizar memoria de la blockchain (y el costo que esto supone) para el mapping de owners y el de auditors. Para este problema en particular no necesitábamos hacer nada con informaciones sobre owners o auditors y no debíamos guardar la información de los mismos. Con el fin de optimizar, se resolvió de dicha forma:

```
mapping(uint256 => address payable) public proposals;  
mapping(address => bool) public owners;  
mapping(address => Maker) public makers;  
mapping(address => bool) public auditors;
```

- Por otro lado, la información de los makers se guardó en un mapping con el tipo de dato Maker en lugar de un booleano a diferencia de los owners o auditors. Esto es porque para el caso de los makers si nos interesaba guardar la información de los mismos. El tipo de dato Maker guardaba la siguiente información y estaba codificado de la siguiente forma:

```
struct Maker {  
    uint256 id;  
    string name;  
    string country;  
    string passportNumber;  
}
```

- Para cumplir con los requerimientos 8 y 9 se decidió crear un modifier *enableProposal* que verifica que existan en el sistema al menos 3 makers y 2 auditors:

```
modifier enableProposal() {  
    require(makerIdsCounter >= 3, "No hay mas de 3 makers");  
    require(auditorIdsCounter >= 2, "No hay mas de 2 auditors");  
    _;  
}
```

- Propuestas:

ID	DESCRIPCIÓN
1	Solo los owners pueden abrir el periodo de presentación de propuestas
2	Solo los owners pueden cerrar el periodo de presentación de propuestas
3	Para abrir un periodo de presentación de propuestas el sistema debe estar en estado neutro. Ni en periodo de presentación de propuestas ni en periodo de votación
4	Para cerrar un periodo de presentación de propuestas debe cumplirse: <ul style="list-style-type: none"> <li>• Encontrarse abierto el periodo de propuestas</li> <li>• Deben haber sido presentadas al menos 2 propuestas</li> </ul>
5	Cada propuesta debe ser controlada por un contrato inteligente específico el cual será gobernado por el contrato SmartInvestment únicamente
6	Los datos que debe incluir cada propuesta son: <ul style="list-style-type: none"> <li>• Nombre</li> <li>• Descripción</li> <li>• Inversión mínima requerida</li> <li>• Dueño de la propuesta. El maker que la presentó.</li> </ul>
7	Las propuestas deben ser validadas por al menos 1 auditor para poder recibir fondos
8	Los contratos inteligentes de las propuestas no deben poder recibir fondos hasta que no se haya abierto el proceso de votación. Por lo que su balance al inicio de cada proceso de votación debe ser cero.

- En Proposal guardamos las addresses del owner (que en un principio es el SmartInvestment) y la del maker, el creador de la propuesta. Esta información luego se utiliza para que en caso de que sea la propuesta ganadora, la transferencia de la propiedad sea lo más eficiente posible.

```
address payable private _owner;

uint256 public _id;
bool public _isOpen;
bool public _audited;
string public _name;
string public _description;
uint256 public _minimumInvestment;
address payable public _maker;
uint256 public _votes;
```

- La creación de la propuesta está definida en SmartInvestment, y quedó desarrollada de la siguiente forma:

```
function createProposal(string calldata name, string calldata description, uint256 minimumInvestment) external payable isMaker proposalPeriod {
    Proposal newProposal = new Proposal(proposalIdsCounter, false, false, name, description, minimumInvestment, msg.sender);
    address payable newProposalAddress = payable(address(newProposal));
    proposals[proposalIdsCounter] = newProposalAddress;
    proposalIdsCounter++;
}
```

- Proceso de Votación:

ID	DESCRIPCIÓN
1	El periodo de votación debe abrirse automáticamente al cerrarse un periodo de presentación de propuestas
2	Para votar por una propuesta cada votante debe transferir al menos 5 ethers al contrato inteligente de la propuesta. No siendo posible retirar nuevamente el dinero depositado.
3	Los votantes pueden votar tantas veces como deseen por cualquier propuesta que deseen
4	Solo los owners pueden solicitar cerrar un periodo de votación
5	Para cerrar un periodo de votación se deben cumplir los siguientes requisitos: <ul style="list-style-type: none"> <li>• Que la suma de los balances de los contratos de las propuestas hayan alcanzado un mínimo de 50 ethers</li> <li>• Que el cierre sea autorizado por al menos 2 auditors</li> </ul>
6	Al errar el periodo de votación debe pasarse a un estado neutro donde no puedan presentarse nuevas propuestas ni realizar votos sobre las propuestas ya presentadas
7	Al cerrar el periodo de votación el recuento de votos debe realizarse en forma automática sobre la cantidad de ethers recibida por cada contrato inteligente, ganando la propuesta del contrato inteligente que posea mayor balance.
8	En caso de empate se contarán la cantidad de votos recibidos por las propuestas empatadas y quien posea mayor cantidad será la propuesta ganadora
9	Una vez determinado el ganador de forma automática se dispararán las siguientes acciones: <ol style="list-style-type: none"> <li>1. A cada contrato inteligente se le cobrará un 10% de su balance como comisiones por el uso del sistema que debe ser transferida al contrato inteligente SmartInvestment</li> <li>2. El saldo acumulado en todos los contratos inteligentes que pertenezcan a propuestas que no hayan ganado, se transferirán al contrato ganador</li> <li>3. Todos los contratos inteligentes correspondientes a las propuestas no ganadoras serán destruidos.</li> <li>4. La propiedad del contrato de la propuesta ganadora será transferida al owner de la propuesta, de forma que será el único capaz de disponer de los fondos del contrato.</li> <li>5. Debe habilitarse el retirar el dinero depositado solamente por el owner</li> </ol>
10	Al determinar la propuesta ganadora debe emitirse de forma automática un evento que guarde la información de la propuesta ganadora con nombre, dueño e inversión mínima requerida. Debe poderse buscar por nombre de propuesta o dueño de la propuesta.

- El cumplimiento del requerimiento 2 fue realizado en la clase Proposal, donde en el método vote() se verifica que la transferencia de Ethers al momento de votar sea de al menos 5 ethers. No se hace ningún chequeo de quien es el Voter, por lo que un Voter puede votar cuantas veces quiera:

```
function vote() external payable {
    require(msg.value >= 5 ether, "Necesita que sean mas de 5 ethers");
    require(_audited, "Proposal no esta auditada");
    require(SmartInvestment(_owner).isVotingPeriod(), "No es periodo de votacion");
    require(SmartInvestment(_owner).isVoter(msg.sender), "No es votante");
    _votes++;
}
```

- La mayoría de los requerimientos relacionados al cierre de el periodo de votación están desarrollados en el método `setStateClosed()`, que fue codificado de la siguiente manera:

```
function setStateClosed() internal {
    uint256 totalBalance = 0;
    for(uint256 i = 1; i < proposalIdsCounter && totalBalance <= 50 ether; i++) {
        totalBalance += address(proposals[i]).balance;
    }
    require(totalBalance >= 50 ether, "Se necesita que el total del balance de las propuestas sea mayor a 50");
    bool auditorsAuthorization = votingCloseAuthorizationAuditors[0] != address(0) && votingCloseAuthorizationAuditors[1] != address(0);
    require(auditorsAuthorization, "Se necesita autorizacion de auditores");
    systemState = SystemState.Closed;
    delete votingCloseAuthorizationAuditors[0];
    delete votingCloseAuthorizationAuditors[1];
    address proposalWinner = getProposalWinner();
    Proposal(proposalWinner).transferTenPercent();
    uint256 proposalWinnerId;
    for(uint256 i = 1; i < proposalIdsCounter; i++){
        if(proposals[i] != proposalWinner){
            Proposal(proposals[i]).transferTenPercent();
            Proposal(proposals[i]).transferFundsAndSelfDestroy(proposalWinner);
            delete proposals[i];
        }
        else {
            proposalWinnerId = i;
        }
    }
    Proposal proposalWinnerObject = Proposal(proposalWinner);
    emit DeclareWinningProposalEvent(proposalWinner, proposalWinnerObject._maker(), proposalWinnerObject._minimumInvestment());
    proposalWinnerObject.transferPropertyToMaker();
    delete proposals[proposalWinnerId];
    proposalIdsCounter = 1;
}
```

- Todas las operaciones relacionadas a las acciones a ser realizadas en las proposals luego de determinado el ganador son dentro del segundo método iterativo `for`, dejando por fuera la proposal ganadora para asegurarnos de que no realice ninguna operación en la misma. Estas operaciones son realizadas previa o posteriormente.
- Dentro del método se verifica que los auditores que aprobaron el cierre del periodo de votación no sean el mismo, para cumplir con el requerimiento 5:

```
function authorizeCloseVoting() external isAuditor {
    if (votingCloseAuthorizationAuditors.length == 0) {
        votingCloseAuthorizationAuditors.push(msg.sender);
    } else {
        if (votingCloseAuthorizationAuditors[0] != msg.sender) {
            votingCloseAuthorizationAuditors.push(msg.sender);
        } else {
            revert();
        }
    }
}
```



Además, se cumplieron las condiciones generales impuestas en la letra:

- Todas las operaciones son ejecutadas por medio de comandos al contrato inteligente SmartInvestment, salvo la de transferir ethers al contrato, que es ejecutada por medio de comandos al contrato inteligente de la propuesta directamente. Esto es logrado gracias al modificador isOwner() en Proposal, donde el se chequea que el msg.sender sea la misma address que el owner (el contrato SmartInvestment, que deploya los contratos de las Proposals).

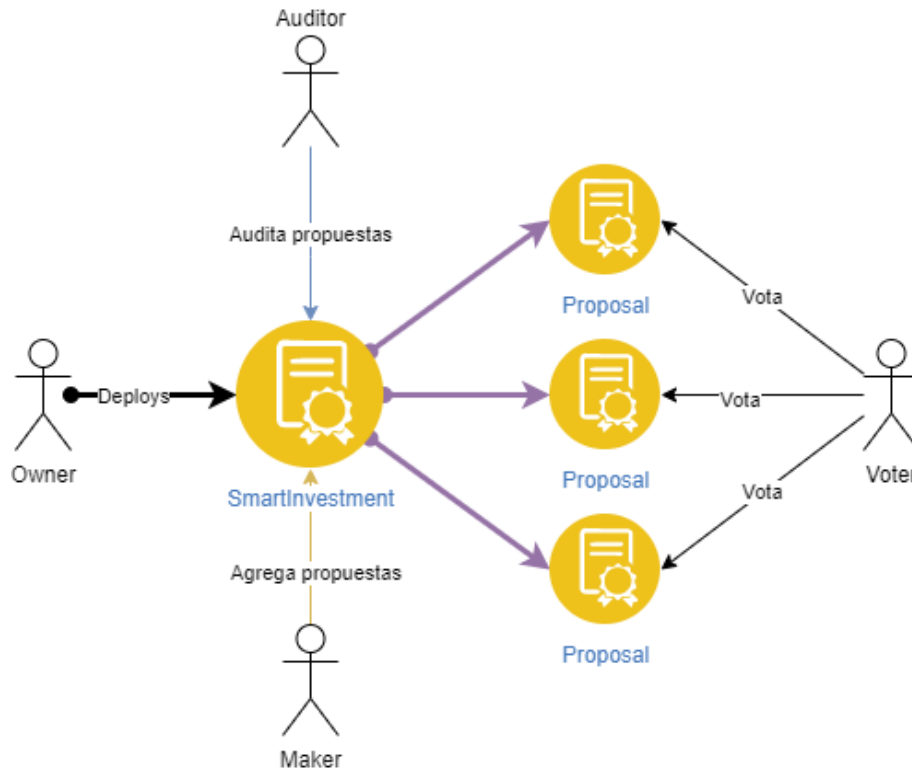
```
5
6 contract Proposal {
7
8     address payable private _owner;
9
10    uint256 public _id;
11    bool public _isOpen;
12    bool public _audited;
13    string public _name;
14    string public _description;
15    uint256 public _minimumInvestment;
16    address payable public _maker;
17    uint256 public _votes;
18
19    constructor(uint256 id, bool isOpen, bool audited, string memory name, string memory description, uint256 minimumInvestment, address maker){
20        _owner = payable(msg.sender);
21        _id = id;
22        _isOpen = isOpen;
23        _audited = audited;
24        _name = name;
25        _description = description;
26        _minimumInvestment = minimumInvestment;
27        _maker = payable(maker);
28    }
29
30    modifier isOwner() {
31        require(_owner == msg.sender, "No es el owner");
32        _;
33    }
34
35    function transferPropertyToMaker() external isOwner {
36        _owner = _maker;
37    }
38
39    function transferFundsAndSelfDestroy(address destinationAddress) payable external isOwner {
```

Lo resaltado en rojo es la definición y seteo del valor de la variable de tipo address \_owner, mientras que en blanco se ejemplifica el uso del modifier.

- Ninguna operación es ejecutada directamente sobre los contratos inteligentes de las propuestas, salvo la transferencia de ethers a la propuesta al momento de votar
- Solo los owners pueden retirar el saldo en ethers del contrato SmartInvestment

# Arquitectura del proyecto

Podemos encontrar la definición de dos contratos: SmartInvestment y Proposal, e interactúan de la siguiente forma con los roles definidos:



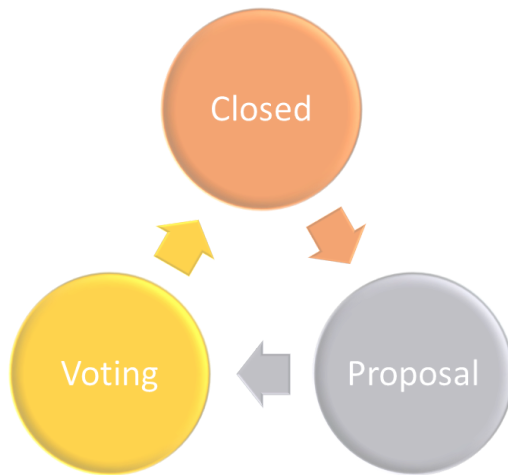
Los owners despliegan el contrato principal SmartInvestment, y pueden agregar más contratos.

Los makers agregan propuestas a través de un llamado a createProposal en SmartInvestment.

Los auditores auditan propuestas a través de un llamado a validateProposal con el id de la propuesta en SmartInvestment, además de autorizar el cierre de la votación.

Los votantes votan por las propuestas haciendo un llamado a vote por la proposal que quieran votar.

## Manejo de estados del sistema



Debido a la posible complejidad que se puede incurrir en el manejo de estados, nos propusimos como objetivo evitar el rompimiento del ciclo de estados (pasar de proposal a closed por ejemplo) por un owner.

Para esto, utilizamos funciones internas para manejar el establecimiento de estados en SmartInvestment, y una función pública externa a la cual podemos llamar para cambiar al próximo estado de nombre switchState. Esta función se encarga exclusivamente de consultar el estado actual y llamar a una función interna con el estado próximo que le corresponda.

Se utilizaron los elementos característicos del lenguaje Solidity para la codificación de contratos inteligentes, ejemplificando a continuación:

- Modifiers

- Ya existentes (pure, view):

```
function getProposalWinner() internal view returns(address){
    address winner = proposals[1];
    for (uint256 i=2; i < proposalIdsCounter; i++) {
        if (proposals[i].balance > winner.balance) {
            winner = proposals[i];
        } else if (proposals[i].balance == winner.balance && Proposal(proposals[i])._votes() > Proposal(winner)._votes()) {
            winner = proposals[i];
        }
    }
    return winner;
}
```

- Creados para cumplir con las reglas del negocio:

```
modifier votingPeriod() {
    require(systemState == SystemState.Voting, "No se encuentra en voting period");
    _;
}
```

- Tipos de variables (en particular el uso de external para los métodos que solo pueden ser llamados desde fuera del contrato):

```
function vote() external payable {
    require(msg.value >= 5 ether, "Necesita que sean mas de 5 ethers");
    require(_audited, "Proposal no esta auditada");
    require(SmartInvestment(_owner).isVotingPeriod(), "No es periodo de votacion");
    require(SmartInvestment(_owner).isVoter(msg.sender), "No es votante");
    _votes++;
}
```

- Eventos:

```
event DeclareWinningProposalEvent(address proposalAddress, address maker, uint256 minimumInvestment);

emit DeclareWinningProposalEvent(proposalWinner, proposalWinnerObject._maker(), proposalWinnerObject._minimumInvestment());
```

- Mappings y addresses:

```
mapping(uint256 => address payable) public proposals;
mapping(address => bool) public owners;
mapping(address => Maker) public makers;
mapping(address => bool) public auditors;
```

- Calldata (fue usado en parámetros de funciones. Usamos calldata y no memory para no ocupar espacio de almacenamiento en la blockchain y el gasto que esto conlleva. Memory tiene costo extra):

```
function createProposal(string calldata name, string calldata description, uint256 minimumInvestment) external payable isMaker proposalPeriod {
    Proposal newProposal = new Proposal(proposalIdsCounter, false, false, name, description, minimumInvestment, msg.sender);
    address payable newProposalAddress = payable(address(newProposal));
    proposals[proposalIdsCounter] = newProposalAddress;
    proposalIdsCounter++;
}
```

- Uso de la interfaz de Proposal (para llamar a los métodos, por ejemplo de Proposal):

```
function validateProposal(uint256 proposalId) external isAuditor {
    if (proposals[proposalId] != address(0)) {
        Proposal(proposals[proposalId]).setAudited();
    }
}
```

- Payable (la función vote() recibirá ethers):

```
function vote() external payable {
    require(msg.value >= 5 ether, "Necesita que sean mas de 5 ethers");
    require(!_audited, "Proposal no esta auditada");
    require(SmartInvestment(_owner).isVotingPeriod(), "No es periodo de votacion");
    require(SmartInvestment(_owner).isVoter(msg.sender), "No es votante");
    _votes++;
}
```

- Selfdestruct para la destrucción del contrato y transferencia de fondos:

```
function transferFundsAndSelfDestroy(address destinationAddress) payable external isOwner {
    selfdestruct(payable(destinationAddress));
}
```

- Revert (en lugar de assert ya que assert consume todo el gas de la operación):

```
function switchState() external isOwner {
    if (systemState == SystemState.Proposal) {
        setStateVoting();
    } else if (systemState == SystemState.Closed) {
        setStateProposal();
    } else if (systemState == SystemState.Voting) {
        setStateClosed();
    } else {
        revert();
    }
}
```

- msg:

```

constructor() {
    owners[msg.sender] = true;
    systemState = SystemState.Closed;
}

function vote() external payable {
    require(msg.value >= 5 ether, "Necesita que sean mas de 5 ethers");
    require(!_audited, "Proposal no esta auditada");
    require(SmartInvestment(_owner).isVotingPeriod(), "No es periodo de votacion");
    require(SmartInvestment(_owner).isVoter(msg.sender), "No es votante");
    _votes++;
}

```

- receive() para SmartInvestment ya que recibirá dinero externamente:

```

contract SmartInvestment {

    receive() external payable {}
}

```

- Uso de send en lugar de transfer():

```

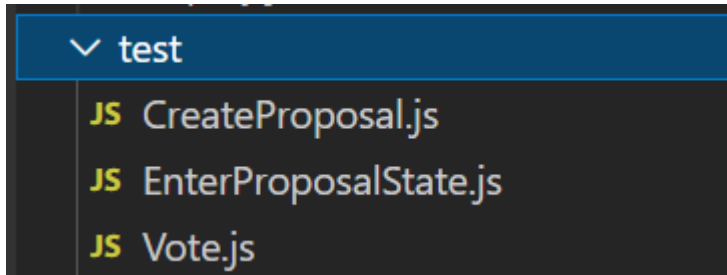
function transferTenPercent() external isOwner {
    uint256 tenPercent = (address(this).balance/10);
    bool transfered = payable(_owner).send(tenPercent);
    if (!transfered) {
        revert("No fueron transferidos!");
    }
}

```

Usamos send en lugar de transfer ya que nos permite obtener un booleano que confirma la transferencia. En caso de que sea fallida, podemos revertir la transacción con un mensaje de error personalizado. Si usamos transfer no podemos mostrar la razón del error, ya que la transferencia lanzaría un error genérico, y sería más difícil testear y debuggear el contrato.

# Testing

Para cumplir con la letra de esta entrega, se procedieron a realizar testings unitarios sobre 3 funcionalidades principales del proyecto en el lenguaje JavaScript, los cuales se encuentran en un directorio en la raíz del proyecto. De estas funcionalidades se testeo gradualmente todos los casos borde (por ej.: En vote que no sea ni maker, ni owner, ni auditor), solucionando un caso borde que falle a la vez (a propósito), hasta llegar a un test satisfactorio. Estos testings son:



Se eligieron estas funcionalidades dado que eran las más centrales del proyecto y/o porque contaban con la mayor cantidad de validaciones, las cuales crean casos borde de fallos que deben suceder bajo determinadas circunstancias.

Para comprobar el cumplimiento del test se utilizó la función “expect()” de la librería “chai”.

```
const { expect, assert } = require("chai");
```

```
expect(contractInstance).to.be.ok;
```

```
try {
  await proposal.connect(addr6).vote();
  expect.fail("Should have thrown exception 'No hay almenos 5 ethers enviados'");
}
catch (error) {
  expect(error.message).to.contain("Necesita que sean mas de 5 ethers");
}
```

Para iniciar el testing se utilizó el comando “`npx hardhat test`” en consola y se obtuvieron los siguientes resultados:

```
PS C:\Users\Alfo\Desktop\ORT-OblitBCI> npx hardhat test

=====DEPLOY TEST===== Create Proposal
  ✓ SmartInvestment deployed

Create proposal
  ✓ Should fail because address is not maker (80ms)
  ✓ Should fail because proposal period not set (75ms)
  ✓ Should succeed (230ms)

=====DEPLOY TEST===== Proposal State
  ✓ SmartInvestment deployed

Switch to proposal state
  ✓ should fail to switch without enough makers (70ms)
  ✓ should fail without enough auditors (110ms)
  ✓ should succeed (111ms)

=====DEPLOY TEST===== Vote
  ✓ SmartInvestment deployed

Vote
  ✓ should fail because there isn't 5 ethers or more (175ms)
  ✓ should fail without proposal being audited (152ms)
  ✓ should fail because it isn't in voting period (179ms)
  ✓ should fail because it isn't a voter (262ms)
  ✓ should succeed (204ms)

14 passing (5s)

PS C:\Users\Alfo\Desktop\ORT-OblitBCI> |
```

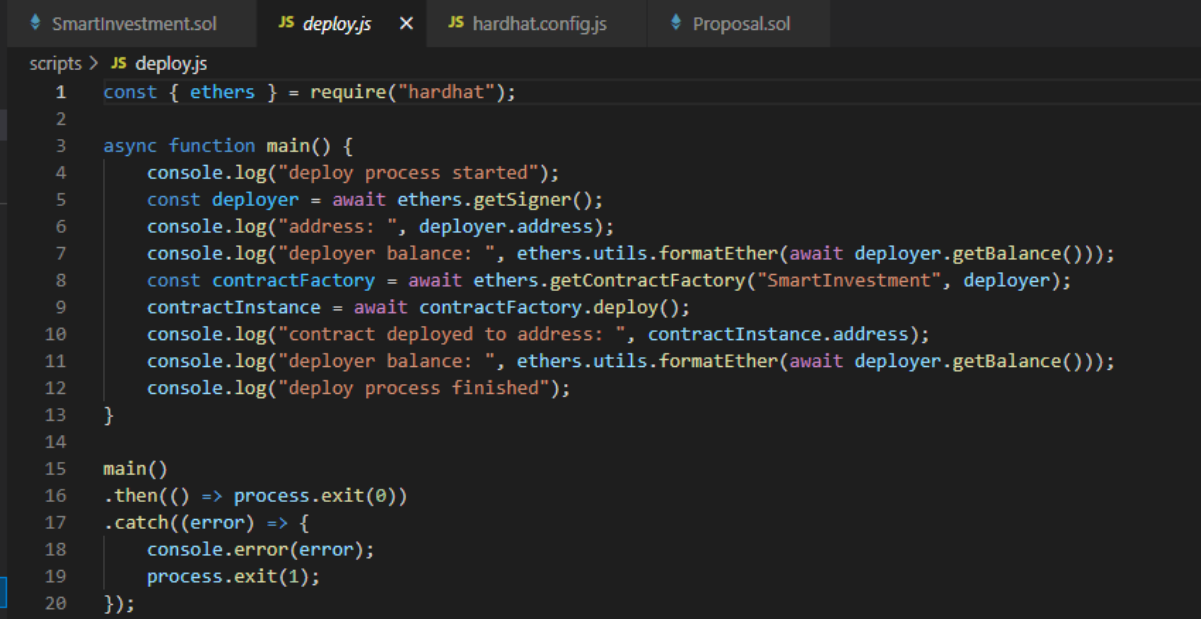
# Deploy

Para el deploy usamos Alchemy y Metamask para la cuenta y la obtención de la clave privada junto con la guía de hardhat.

<https://dashboard.alchemyapi.io/apps/eh19m798kkx3ypmp>  
<https://hardhat.org/tutorial/deploying-to-a-live-network.html>

Para realizar el deploy usamos el comando:

```
npx hardhat run ./scripts/deploy.js --network ropsten
```

A screenshot of a code editor with a dark theme. The editor has four tabs at the top: 'SmartInvestment.sol', 'JS deploy.js' (active), 'JS hardhat.config.js', and 'Proposal.sol'. The main area shows the content of 'deploy.js'. The code is as follows:

```
scripts > JS deploy.js
1  const { ethers } = require("hardhat");
2
3  async function main() {
4      console.log("deploy process started");
5      const deployer = await ethers.getSigner();
6      console.log("address: ", deployer.address);
7      console.log("deployer balance: ", ethers.utils.formatEther(await deployer.getBalance()));
8      const contractFactory = await ethers.getContractFactory("SmartInvestment", deployer);
9      contractInstance = await contractFactory.deploy();
10     console.log("contract deployed to address: ", contractInstance.address);
11     console.log("deployer balance: ", ethers.utils.formatEther(await deployer.getBalance()));
12     console.log("deploy process finished");
13 }
14
15 main()
16   .then(() => process.exit(0))
17   .catch((error) => {
18       console.error(error);
19       process.exit(1);
20   });
```

El archivo deploy.js permite establecer los parámetros para iniciar el deploy en la network de preferencia.

Además logueamos a consola los pasos del script de deploy para ver en qué estado se encuentra y a qué dirección se deployo.



```

hardhat.config.js
1 require("@nomiclabs/hardhat-waffle");
2 require("dotenv").config();
3
4 require("@nomiclabs/hardhat-ethers");
5
6 const ALCHEMY_API_KEY = "-LI-bHpSVf1Jf_fGUEyxGTXdgeZ2ww6E";
7
8 const ROPSTEN_PRIVATE_KEY = "6bde2cc1f91168cc6298a0fcf4e1405d56c70a4893daf2f99ab9173f8";
9
10 /**
11  * @type import('hardhat/config').HardhatUserConfig
12  */
13 module.exports = {
14   solidity: "0.8.4",
15   networks: {
16     ropsten: {
17       url: `https://eth-ropsten.alchemyapi.io/v2/${ALCHEMY_API_KEY}`,
18       accounts: [`${ROPSTEN_PRIVATE_KEY}`]
19     }
20   }
21 };

```

```

PS C:\Users\Alfo\Desktop\ORT-ObliTBCI> npx hardhat run scripts/deploy.js --network ropsten
deploy process started
address: 0x6B7589F276aee865d7DeFEe70751F196379CE742
deployer balance: 0.3
contract deployed to address: 0xbaE90f7aF81cF1313De8860141D7dAd4dFb1C628
deployer balance: 0.3
deploy process finished
PS C:\Users\Alfo\Desktop\ORT-ObliTBCI>

```

Para deployar el contrato elegimos utilizar la red de testeo Ropsten y los faucets de ethereum que usamos son de los siguientes links:

<https://faucet.dimensions.network/>

<https://faucet.ropsten.be/>

Esto nos permitió obtener ETH en la cuenta gestionada por la wallet de Metamask, en la cual agregamos ethers para que nos permita deployar el contrato.

En la siguiente imagen se muestra los movimientos de la cuenta que usamos:

0x6B7589F276aee865d7DeFEe70751F196379CE742

**Etherscan**  
Ropsten Testnet Network

Address: 0x6B7589F276aee865d7DeFEe70751F196379CE742

**Overview**

Balance: 5.287218290995751188 Ether

**More Info**

My Name Tag: Not Available

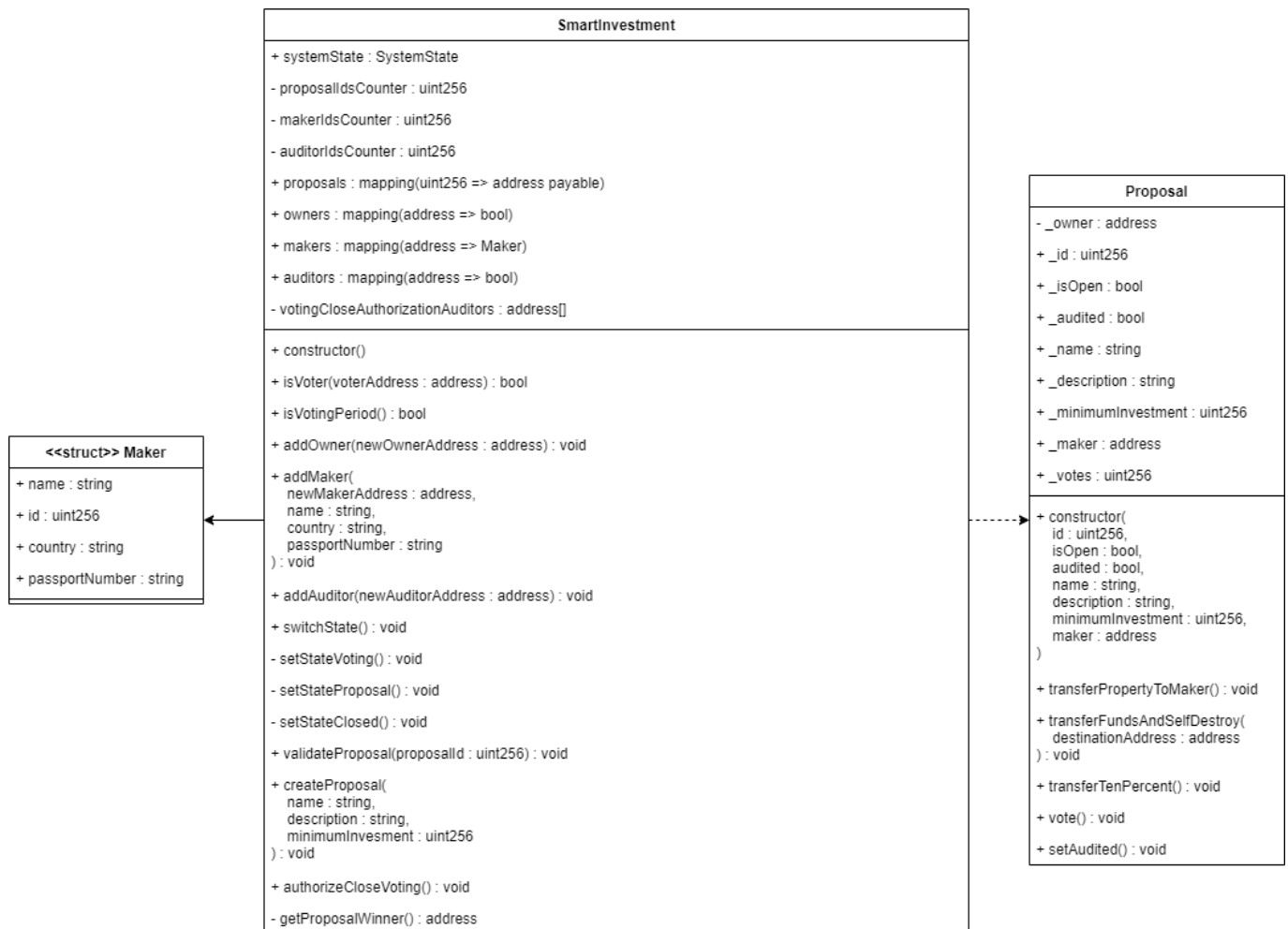
**Transactions**

Latest 3 from a total of 3 transactions

Txn Hash	Method	Block	Age	From	To	Value	Txn Fee
0xb6195da695868bc571...	Transfer	11529604	5 days 18 hrs ago	0x78c115f1c8b7d0804fb...	IN 0x6b7589f276aee865d7...	5 Ether	0.00105
0x7ecab3620384b2f57c...	0x60806040	11529574	5 days 18 hrs ago	0x6b7589f276aee865d7...	OUT Contract Creation	0 Ether	0.012781709004
0x263a222c4c35971f49...	Transfer	11529523	5 days 18 hrs ago	0xcdad06adcd0f1ccea67...	IN 0x6b7589f276aee865d7...	0.3 Ether	0.0000525

[ Download CSV Export ]

# Diagrama de Contratos



Se realizaron dos contratos Proposal y SmartInvestment.

## Contrato SmartInvestment

SmartInvestment es el contrato del sistema principal, y contiene a las propuestas con las direcciones de los diferentes roles, así como la lógica del sistema principal. Es pertinente aclarar que el owner del contrato se asigna en el constructor, con la dirección que realiza el deploy del contrato.

## Administración de roles

Contiene dos mappings de Owners y Auditors, que nos permiten conocer si la dirección se encuentra asignada a ese rol. El mapping de Maker contiene como valor la referencia a un struct debido a que resultó necesario guardar datos extra sobre el Maker como el país y el número de pasaporte. Estos mappings son utilizados por modificadores que permiten restringir el acceso de los métodos a roles determinados.

## Métodos de la lógica

- **switchState**: Es quien maneja el cambio de estados, los cuales pueden ser “closed”, “proposal” y “voting”, siendo estos alternados de forma cíclica por los owners al llamar a dicho método. El estado actual se guarda en el atributo `systemState`, y es de tipo enumerado.
- **validateProposal**: Es el método que permite que un auditor valide una propuesta.
- **authorizeCloseVoting**: Permite conceder la autorización para el cierre del período de votación, la cual debe ser autorizada por dos auditores diferentes, y es el paso previo al proceso de cálculo y asignación de la propuesta ganadora.

El resto de los métodos de acceso público son auto descriptivos, su accionar puede ser inferido a partir del nombre.

## Contrato Proposal

Proposal contiene los atributos que describen propiedades del contrato, así como la cantidad de votos.

### Manejo de la propiedad de la propuesta

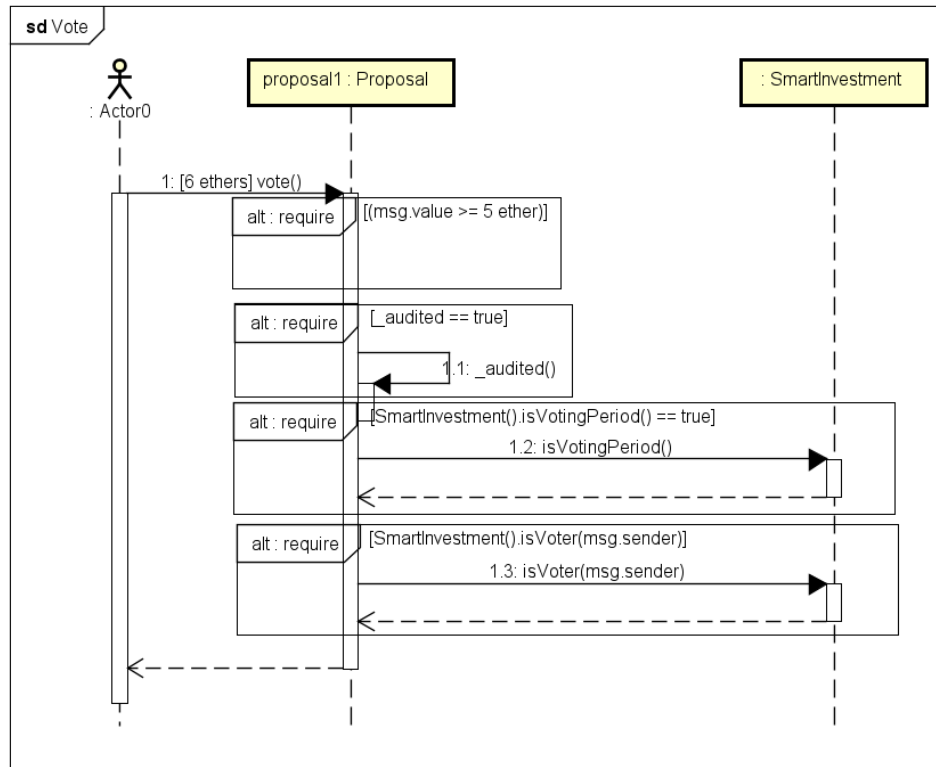
Dos atributos de relevancia son `_maker` y `_owner`. Durante la creación del contrato, el cual es creado por `SmartInvestment` cuando se hace un llamado a `createProposal`, se asigna su dirección a `_owner`, y la dirección del maker (quien llamó a `createProposal`) a `_maker`.

Una vez que la propiedad del contrato ganador se transfiere cuando el owner llama a `transferPropertyToMaker`, el maker pasa a ser owner del contrato, y la dirección de atributo `_maker` se copia a `_owner`.

### Lógica de votación

El contrato contiene un método `vote` el cual consulta al contrato principal `SmartInvestment` si nos encontramos en período de votación, y si quien realiza la llamada es del tipo votante, además de requerir que la cantidad de ethers enviada sea mayor a 5 y el contrato se encuentre auditado.

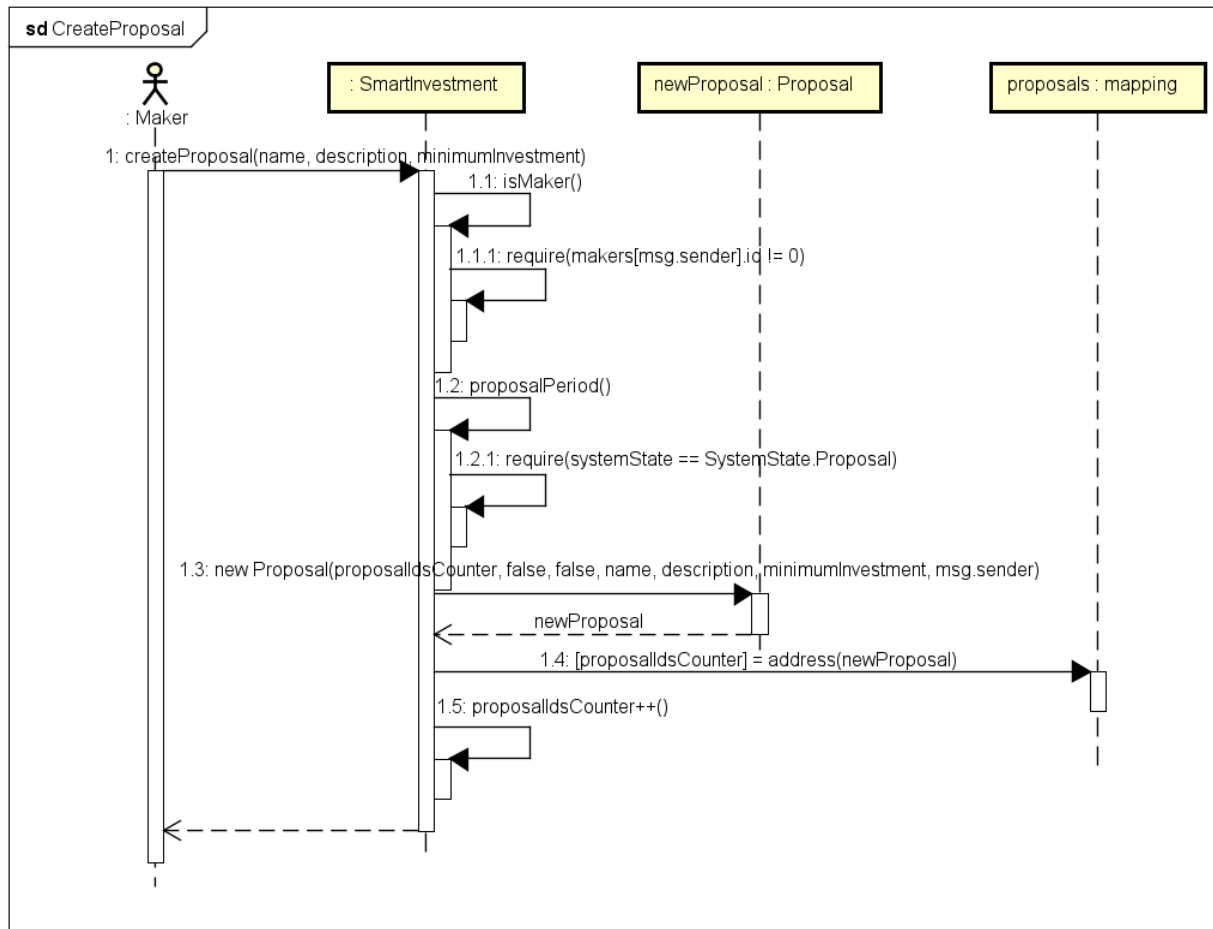
# Diagramas de Secuencia



El proceso de votación consta de que un votante sobre el address de la proposal que desea votar, envíe la cantidad de ethers (mayor a 5) que desee. Dentro de la función de voto se itera sobre 4 “require” que revisan:

1. Que el votante haya enviado al menos 5 ethers.
2. Que la proposal haya sido auditada por un “Auditor”.
3. Que se esté en periodo de votación (votingPeriod). Que lo chequea con la función isVotingPeriod del contrato principal “SmartInvestment”.
4. Que el votante sea un votante, osea que no sea Owner, ni auditor, ni maker.

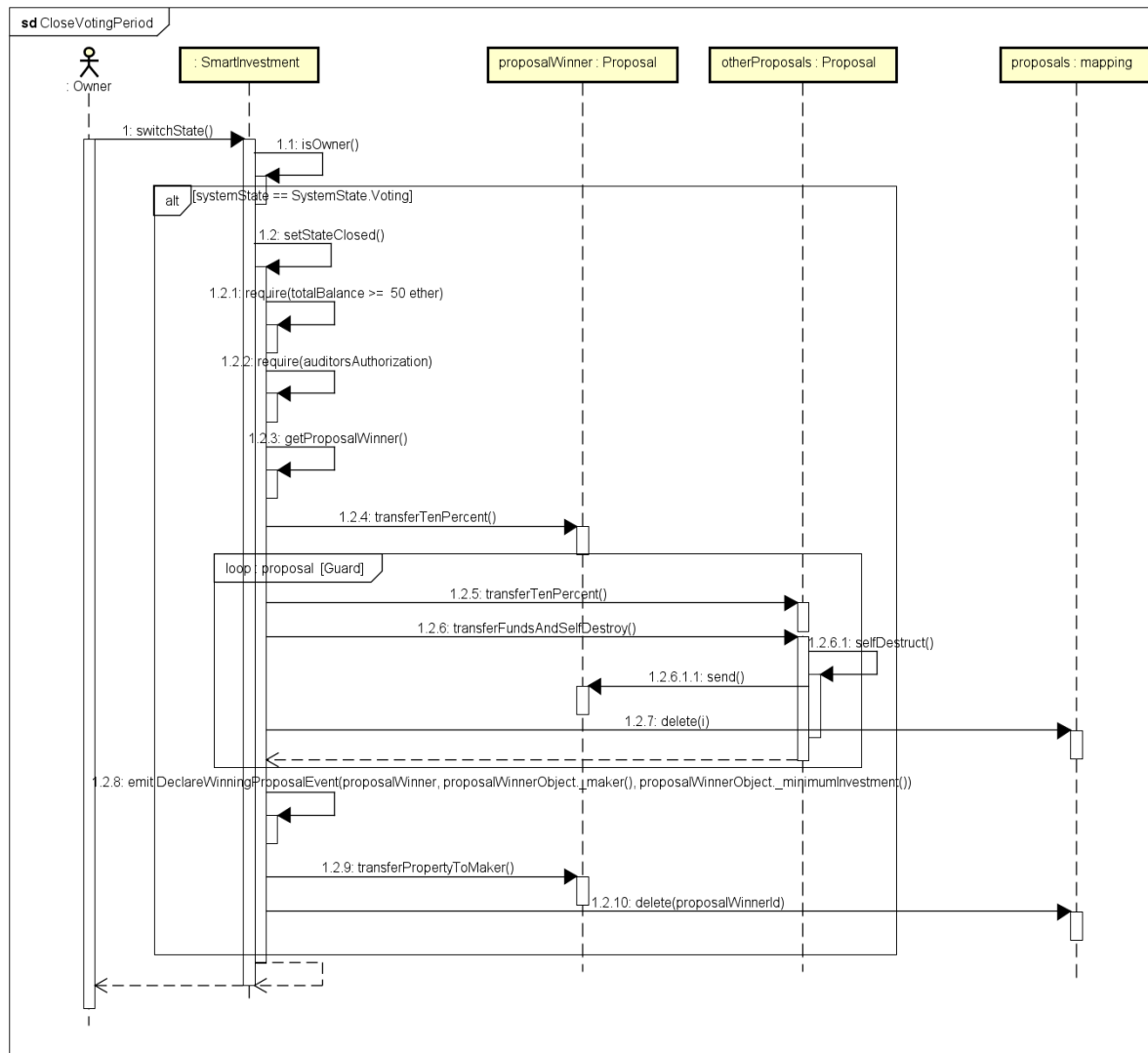
Si todos estos requieren se cumplen, acepta el voto. Si alguno falla hace un “revert”.



CreateProposal se encarga de registrar en el mapping del contrato principal el contrato luego de pasar por ciertas verificaciones. Lo primero es que el que esté llamando a CreateProposal sea un maker y su id sea distinto de cero para indicar que no es el owner. Luego se verifica que se esté en ProposalPeriod, la cual es una función que tiene un require que verifica que el estado del sistema sea igual al estado de sistema de Proposal. Si estos requerimientos previamente mencionados cumplen, se pasa a crear la proposal con la información enviada en el llamado a la función CreateProposal y en la posición en el mapping del id de la nueva proposal se guarda la proposal y se aumenta en 1 el proposalsIdsCounter.

CloseVotingPeriod es el proceso de llamar a SwitchState, estando el sistema en estado de votación, lo cual ocasiona que se pase el estado del sistema de votación a cerrado. Para cambiar el estado se tiene que ser un Owner, si es un owner entra en una iteración de If's no

contemplada en el diagrama, hasta llegar al `if(systemState==SystemState.Voting)`, que si se cumple llama al método interno `setStateClosed()` que se encargará de realizar las verificaciones de requerimientos previas al cierre.



Para empezar verifica con un `for` (no contemplado por la baja relevancia de la manera de hallar el `balanceTotal` de todas las `proposals`). Al salir del `for`, un `require` verifica que el balance total sea mayor o igual a 50 ethers. Luego se verifica con otro `require` que se tenga la autorización del auditor. Si estos se cumplen, se llama a `getProposalWinner` para hallar cuál fue la propuesta que ganó y llamar al método `transferTenPercent` dentro de la `proposalWinner` para recibir el 10% de comisión. Luego se procede a realizar el proceso de transferencia del 10% de comisión y la transferencia de fondos restantes al `proposalWinner`, autodestruir todos las `proposals` perdedoras y eliminarlas del `mapping` desde dentro de un `for`. Una vez finalizado este proceso se emite el evento de Declarar la Proposal Ganadora del evento, transferir la propiedad de la propuesta ganadora al `maker` y borrar por último está `proposal` del `mapping` de `proposals`.

## Posibles mejoras y conclusiones

A lo largo del código hemos utilizado las funciones de `require` y `revert` al verificar que se cumplan las condiciones de la letra. Estas funciones tienen mensajes de error en español para los casos donde no se cumpla la condición del `require` o se ejecute el `revert`. Como futura mejora, se debe ser consistente y que dichos mensajes de error estén en inglés.

Si bien era una funcionalidad opcional, para esta entrega no implementamos el patrón Proxy para poder actualizar el contrato inteligente en caso de que sea necesario arreglar un bug o modificar las reglas del negocio. Consideramos que hubiese sido oportuno hacerlo ya que en una eventual codificación de un proyecto con un cliente real es un requerimiento que seguramente esté presente y sería interesante ya haber tenido una experiencia en la aplicación del patrón.

De la misma forma, otra funcionalidad que no era obligatoria y no cumplimos fue darle la posibilidad a los owners de pausar el sistema en caso de una situación de emergencia. El patrón pausable es un patrón que se utiliza en casi todos los proyectos principalmente por razones de seguridad. Permite pausar el contrato cuando sea necesario en caso de un hackeo o un problema de seguridad, con el objetivo de que se pueda arreglar el bug en el código desde donde se origina el problema y así poder continuar con el negocio.

Por último, al seguir las guías presentadas previamente del deploy, no guardamos las constantes en un archivo separado `.env`, sino que están declaradas en el script de configuración (`hardhat.config.js`). Para perfeccionar el proyecto, estas constantes deberían estar declaradas en un archivo aparte `.env`.