МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА СИСТЕМИ ЕЛЕКТРОННОГО ГОЛОСУВАННЯ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ БЛОКЧЕЙН»**

Виконав: студент 4 курсу, групи 6.1219-1пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
В.О. Звонарьов
(ініціали та прізвище)

Керівник завідувач кафедри програмної інженерії, доцент, к.ф.-м.н. Лісняк А.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

« 07 » 02 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Звонарьову Владиславу Олеговичу
(прізвище, ім'я та по-батькові)

1. Тема роботи Розробка системи електронного голосування з використанням технології блокчейн

керівник роботи Лісняк Андрій Олександрович, к.ф.-м.н., доцент
(прізвище, ім'я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 26 » січня 2023 року № 102-с

2. Строк подання студентом роботи 07.06.2023 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі, аналіз предметної області.
2. Проєктування програмного забезпечення.
3. Реалізація та тестування програмного доповнення.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
презентація за темою докладу

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 07.02.2023 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану виконання кваліфікаційної	08.02.2023	
	роботи бакалавра.		
2.	Збір вихідних даних та аналіз предметної	22.02.2023	
	області.		
3.	Обробка методичних та теоретичних джерел.	15.03.2023	
4.	Робота над першим розділом.	05.04.2023	
5.	Робота над другим розділом.	26.04.2023	
6.	Робота над третім розділом.	10.05.2023	
7.	Робота над додатками.	17.05.2023	
8.	Оформлення та нормоконтроль	01.06.2023	
	кваліфікаційної роботи бакалавра.		
9.	Захист кваліфікаційної роботи бакалавра.	21.06.2023	

Студент _____ В.О. Звонарьов
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.О. Лісняк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка системи електронного голосування з використанням технології блокчейн»: 64 с., 43 рис., 24 джерела, 3 додатки.

CHAI, HURDHAT, JAVASCRIPT, RINCEBY, SOLIDITY.

Об’єкт дослідження – взаемодія з blockchain, інструменти для створення та тестування смарт-контрактів.

Мета роботи: розробити систему голосування з використанням технології blockchain.

Метод дослідження – моделювання, проєктування, програмний, аналітичний.

Останнім часом технології у сфері фінансів зробили величезний стрибок уперед, ми вже звикли оплачувати більшість наших платежів за квартиру, продукти та розваги онлайн. З огляду на те, що люди завжди будуть бажати якомога зручніше й ефективніше взаємодіяти з грошима, я вибрав сферу взаємодії з криптовалютою як свою дипломну роботу.

Однією з переваг моєї роботи є надійність за допомогою використання деяких патернів, які ми розглянемо пізніше, та економія газу через дотримання правил, про які ми можемо дізнатися, прочитавши документацію Ethereum.

Окрім цього, ми розглянемо такі поняття як L0, L1, L2, L3 блокчейни, взаємодії з ними, познайомимося з основними інструментами Solidity розробників, а також вивчимо основні види атак на смарт-контракти і способи їх запобігання..

Таким чином, за результатами роботи створено систему голосування та розглянуті нові технології для взаємодії з фінансовим ринком багатьох країн.

SUMMARY

Bachelor’s qualifying paper «Development of an Electronic Voting System using Blockchain Technology»: 64 p., 43 figures, 24 references, 3 supplements.

CHAI, HURDHAT, JAVASCRIPT, RINCEBY, SOLIDITY.

Object of research is interaction with blockchain, tools for creating and testing smart contracts.

The aim of the study is to develop a voting system using blockchain technology.

Research methods are modeling, design, software, analytical.

Recently, technologies in the field of finance have made a huge leap forward, we are already used to paying most of our rent, food and entertainment payments online. Given that people will always want to interact with money as conveniently and efficiently as possible, I chose the area of cryptocurrency interaction as my thesis.

One of the advantages of my work is reliability through the use of some patterns, which we will discuss later, and gas savings through following the rules, which we can learn about by reading the Ethereum documentation.

In addition, we will look at such concepts as L0, L1, L2, L3 blockchains, interactions with them, get acquainted with the main Solidity tools of developers, and learn the main types of attacks on smart contracts and ways to prevent them.

Thus, the results of the work include the creation of a voting system and the consideration of new technologies for interaction with the financial market of many countries.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Технічне завдання 9
1.1 Технологія Blockchain 9
1.2 Різноманітності нод 10
1.3 L0, L1, L2, L3, L4 блокчейни 13
1.4 Криптовалюта 15
1.5 Порівняння сучасних мов програмування для написання смарт-контрактів 17
1.6 Огляд і вибір інструментів розробки та обгрантування вибору 18
1.7 Технічне завдання 19
1.7.1 Опис предметної області 19
1.7.2 Функціональні вимоги 19
1.7.3 Нефункціональні вимоги 20
1.7.4 Опис основних етапів створення смат-контрактів 20
1.8 Висновки до розділу 1 21
2 Проєктування 23
2.1 Загальне призначення 23
2.2 Створення діаграм 23
2.2.1 Діаграма прецедентів 23
2.2.2 Функціональна діаграма 25
2.2.3 Структурна схема 26
2.3 Аналіз паттернів проєктування у Solidity 27
2.4 Проєктування з урахуванням захисту від різних типів атак 31
2.4.1 Dos атака 31

2.4.2 Reentrancy атака 32
2.5 Висновки до розділу 2 34
3 Створення та розгортання смарт контракту 35
3.1 Встановлення середовища розробки Remix та додаткових бібліотек 35
3.2 Написання смарт-контракту 36
3.3 Встановлення Hardhat і залежностей 42
3.4 Написання тестів для смарт-контракту 44
3.5 Деплой смарт-контракту 46
3.5.1 Деплой у локальну тестову мережу Hardhat 47
3.5.2 Деплой у тестову мережу Goerli 50
3.6 Висновки до розділу 3 53
Висновки 54
Перелік посилань 55
Додаток А Смарт-контракт системи голосування 57
Додаток Б Тести для перевірки роботи контракту 61
Додаток В Скрипт розгортання контракту 64

ВСТУП

Коли ми поринаємо у глибокий світ цифрових фінансів то частіше за все ми чуємо такі дивні слова як криптовалюта. У цій дипломній роботі я хочу зробити уклон не тільки на саму систему голосування, а й освітити такі неявні теми як навіщо взагалі потрібна ця технологія, чим відрізняються різноманітні блокчейни, які функції вони використовують, на яку саме працю програмісти зможуть розраховувати в найближчому майбутньому часі та використання криптовалюти у нашому житті [1].

У технічній частині цієї роботи буде створена система голосування з використанням мови програмування Solidity та JavaScript. Будуть розглянуті патерни проєктування для смартконтрактів та деякі особливості написання для збереження коштів при завантаженні.

Основними завданнями роботи будуть:

- огляд технологічного стека у сфері блокчейн розробки;
- аналіз паттернів і вразливостей під час написання смартконтрактів;
- проєктування смартконтракту з використанням UML діаграм;
- створення системи голосування на базі мови програмування Solidity з використанням патернів захисту від вразливостей;
- інтеграція системи у тестову мережу Goerli.

1 ТЕХНІЧНЕ ЗАВДАННЯ

1.1 Технологія Blockchain

Blockchain – це технологія, яка з'явилася в 2008 році в результаті розвитку криптовалюти Bitcoin. З тих пір вона стала популярною та застосовується в різних галузях, таких як фінанси, медицина, логістика та інші [2].

Blockchain це база даних, яка зберігається на багатьох комп'ютерах, що знаходяться в різних місцях світу та працюють в мережі під керуванням спеціального програмного забезпечення. Кожен комп'ютер, який зберігає реєстр, називається вузлом мережі. Кожен вузол мережі має копію реєстру, яка оновлюється при кожній новій транзакції. Це дозволяє забезпечити безпеку та надійність зберігання даних, а також унеможливити їх несанкціонований доступ чи зміну [3].

Як я вже сказав, децентралізація забезпечується завдяки створенню численних копій бази даних, які фізично зберігаються на різних пристроях і в різних географічних регіонах, але водночас синхронізовані та спільно додають у базу нову інформацію. Цим блокчейн принципово відрізняється від централізованих мереж, що працюють за принципом клієнт-сервер (див. рис. 1.1).

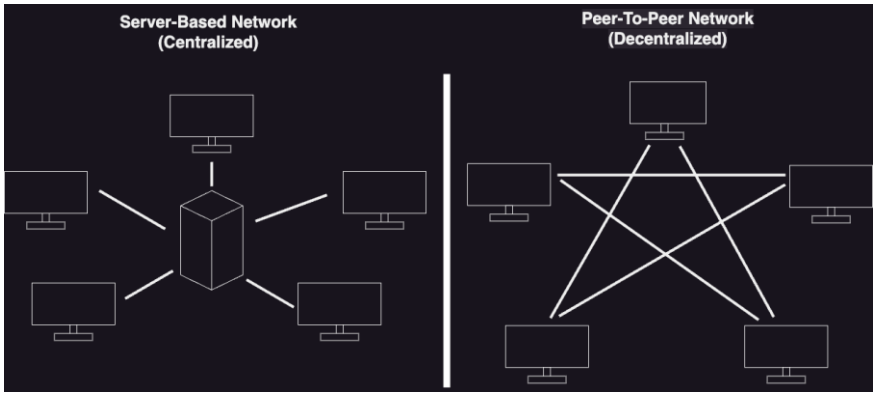

Рисунок 1.1 – Принцип централізованої та децентралізованої мережі

Але уся ця технологія не змоглаб працювати без використання серверів-вузлів, або нод.

1.2 Різноманітності нод

Нодами називають ті самі пристрої, на яких зберігаються копії даних блокчейна і які додають у ланцюжок блоки з новими транзакціями на основі механізму консенсусу. Технічно ноди складаються з [4]:

- обладнання: віртуальний або фізичний сервер + обладнання для майнінгу;
- софту: який дозволяє ноді взаємодіяти з іншими вузлами мережі та правильно виконувати покладені на неї функції.

Блокчейн включає кілька різновидів нод для розв'язання різних завдань, але ядро мережі становить клас повних нод (full nodes), що виконує такі завдання:

- перевірка блоків: за обробку транзакцій і формування блоків відповідають майнери або валідатори, але кожен повний вузол мережі повинен перевірити сформований блок і додати його до своєї копії історії транзакцій або відхилити (саме тому для реалізації хардфорків, що зачіпають базові параметри роботи блокчейна, потрібна підтримка всіх нод – інакше вони просто не зможуть перевіряти транзакції, що відповідають новим правилам);
- зберігання історії транзакцій: ноди записують і зберігають на своїх накопичувачах повну або часткову інформацію про транзакції, що практично виключає безповоротну втрату даних про стан мережі;
- ретрансляція даних: після того, як нода перевіряє і додає до своєї копії бази даних блок, вона може транслювати цю інформацію іншим вузлам, щоб забезпечити синхронізацію стану мережі та цілісність баз даних;

– підключення до блокчейну: всі дії в мережі, чи то ініціалізація транзакції, чи то виклик функції смарт-контракту, проводяться через ноди.

Крім перерахованих вище основних функцій на ноди можуть покладатися додаткові завдання, для яких потрібна спеціальна конфігурація, тому архітектура блокчейна за фактом є мережею взаємопов'язаних нод із різними функціями (див. рис. 1.2).

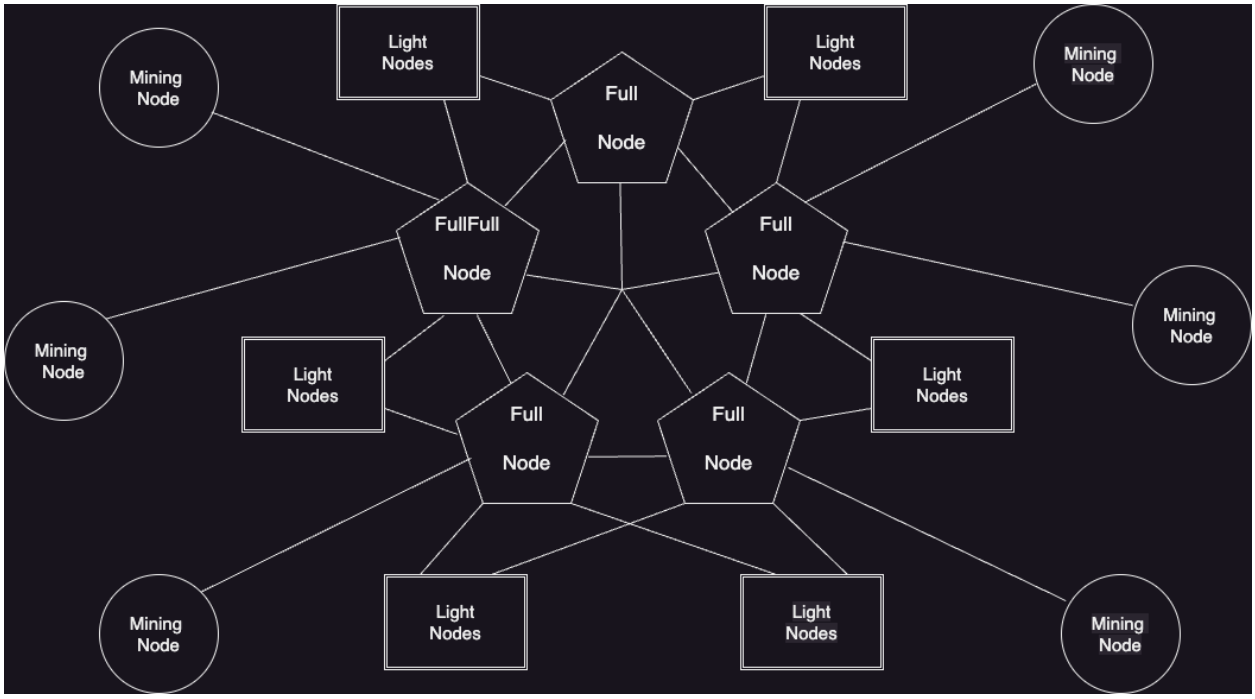

Рисунок 1.2 – Приклад взаємодії нод

Залежно від виконуваних завдань і конфігурації ноди діляться на кілька категорій.

Легкі ноди: зберігають тільки часткові дані про транзакції, тому можуть перевіряти блок самостійно і при синхронізації бази даних змушені покладатися на full nodes. З перерахованих вище базових функцій повноцінно можуть забезпечити тільки доступ до блокчейну.

Повні ноди (full nodes): на відміну від легких нод, зберігають всю історію транзакцій, що розширює можливості для взаємодії з блокчейном. Повні ноди зі свого боку діляться ще на кілька підкатегорій:

а) *урізані повні ноди*: також зберігають історію, але тільки частково, наприклад, останні 20 GB;

б) *архівні повні ноди*: ці вузли зберігають повну історію транзакцій за весь час існування мережі, наявність доступу до повної бази даних дає змогу покласти на архівні повні ноди додаткові функції, тому на їхній основі можуть бути запущені:

– *майнінг ноди*: відповідають за обробку транзакцій і формування блоків у мережах, що працюють на алгоритмі консенсусу PoW (майнінг ноди крім стандартної конфігурації повного вузла також мають обчислювальні потужності, необхідні для обробки та хешування транзакцій, наприклад, ASIC);

– *стейкінг ноди*: виконують ту саму функцію, що й майнінг ноди, але в мережах з алгоритмом консенсусу PoS (технічні вимоги для стейкінг нод можуть бути вищими, ніж для базових повних, але основна відмінність у тому, що для їхнього запуску потрібно застейкати певну суму в нативних токенах мережі);

– *authority ноди*: це аналог валідаторів для мереж з концепцією PoA (VeChain) або PoSA (Binance Smart Chain) (такі ноди допускаються до обробки транзакцій тільки на основі репутації);

– *мастерноди*: на відміну від наведених вище прикладів, вони не формують блоки, але можуть виконувати інші завдання (конкретні права і конфігурація мастернод відрізняються від блокчейна до блокчейна, наприклад, у мережі Dash тільки мастерноди можуть проводити Instant і Private транзакції).

Також виокремлюють 2 категорії нод, що не входять у загальну ієрархію:

– *суперноди*: у різних джерелах класифікується як різновид повних нод або мастернод із додатковим функціоналом;

– *lightning ноди*: спеціальні вузли, що використовуються в рішенні Lightning Network для Bitcoin (lightning ноди потрібні для запуску та

підтримки прямого платіжного каналу між кількома адресами поверх основної мережі Bitcoin).

Варто враховувати, що не всі ноди доступні для вільного розгортання. Якщо повну або стейкінг/майнінг ноду можна розгорнути без дозволу, то для запуску майстер- або authority ноди потрібно відповідати додатковим вимогам і пройти процедуру схвалення.

1.3 L0, L1, L2, L3, L4 блокчейни

Концепція шарів [5] – це категоризація блокчейнів, необхідна для швидкого розуміння, як той чи інший проєкт вписується в екосистему загалом.

Блокчейн-мережі складаються з вузлів (комп'ютерів у мережі, на яких зберігають весь реєстр транзакцій). Це означає, що кожен вузол має задіяти значні обчислювальні ресурси, щоб забезпечити доступ до реєстру та підтримувати консенсус (згоду додавання нового блоку транзакцій до реєстру). Що більш децентралізований блокчейн, то більше в ньому вузлів. Хоча ця надмірність хороша для безпеки мережі, вона погано впливає на її швидкість. Пов'язано це з тим, що багато вузлів беруть участь у перевірці транзакцій. Зі свого боку, мала кількість вузлів підвищить уразливість мережі. Контролюючи 51% вузлів, хакери зможуть керувати мережею як хочуть. Таким чином зробити одночасно ідеально децентралізований, масштабований і безпечний блокчейн наразі не можна, ця проблема відома як трилема блокчейна.

Трилема блокчейна: припущення, що децентралізовані мережі можуть забезпечити тільки дві з трьох переваг у будь-який момент часу щодо децентралізації, безпеки та масштабованості.

До блокчейнів нульового рівня відносять шари які допомогають блокчейнам взаємодіяти один з одним, а саме:

– дають змогу передавати активи між різними блокчейнами;
– дають змогу розробляти один застосунок одразу на кількох блокчейнах (або щонайменше спрощують розробку на кількох блокчейнах через стандартизацію);
– швидкі та дешеві транзакції на кросчейн-біржах, завдяки «комунікаційним» протоколам, що використовуються в L0.

До блокчейнів першого шару відносять блокчейни, які можуть проводити транзакції без участі іншої мережі. Блокчейни першого шару страждають від трилеми блокчейна, і зазвичай добре реалізують лише дві властивості трилеми з трьох (децентралізація, безпека і масштабованість). Для вирішення подібної проблеми існують L2 рішення. Також важливо зазначити, через зростаючу кількість блокчейнів першого рівня, виникає проблема передання активів з одного блокчейна в інший, для цієї проблеми є шар L0.

Шаром другого рівня зазвичай називають сторонні інтеграції з L1, які розв'язують одну з проблем трилеми блокчейну для L1. Найчастіше це проблема масштабування.

Канали станів (*State Channels*) – обмін поза блокчейном транзакціями, після якого в сам блокчейн записується результат (зазвичай реалізується мультипідписним смарт-контрактом).

Вкладений блокчейн (*Nested blockchains*) – робота L2 блокчейна (більш масштабованого, наприклад, за рахунок менш безпечного механізму консенсусу) поверх L1.

Згортки (*Rollups*) – об'єднання декількох транзакції та обробка їх у мережі L1.

Сайдчейни (*Sidechains*) – гібрид вкладеного блокчейна і каналу стану, що використовуються для обробки великої кількості транзакцій разом.

Приклади: Polygon, Optimism.

Шар третього рівня часто називають прикладним шаром. Це рівень, на якому розміщуються децентралізовані додатки та протоколи, що

забезпечують роботу додатків (рис. 1.3).

Приклади: Uniswap, Orbs.

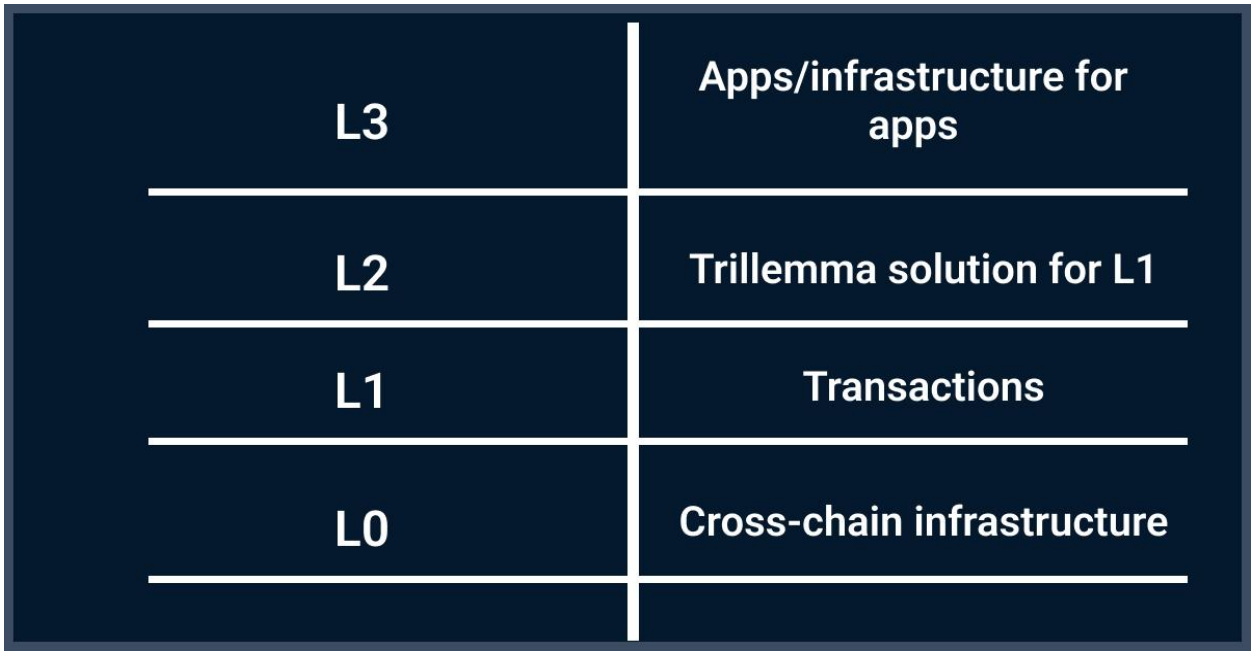

Рисунок 1.3 – Приклади використання різних шарів блокчейну

Звісно, подібна категоризація не завжди зручна, деякі додатки можуть потрапляти в кілька шарів одразу. Наприклад, платформу OmniLayer, створену для торгівлі призначеними для користувача цифровими активами і валютами, побудована поверх Біткойна, можна віднести і до L2, і до L3 шару. Але тим не менш, категоризація за шарами дає змогу прикинути місце проєкту, про який ви чуєте вперше, у блокчейн-екосистемі.

1.4 Криптовалюта

Криптовалюту можна описати по-різному, але в цій дипломній роботі ми розглядатимемо її у вигляді токена зі стандартом ERC20 [6].

ERC-20 розшифровується як Ethereum Request For Comments, число 20 – унікальний ідентифікатор, що відрізняє стандарт від інших. ERC токени – це цифрові активи, розроблені, випущені та використовувані так само, як

біткоїн, за винятком того, що вони працюють виключно на блокчейні Ethereum.

Стандарт ERC-20 було вперше встановлено у 2015 році. У той час створювалися сотні нових монет. Інвесторам і розробникам потрібен був простий спосіб обміну між усіма цими різноманітними монетами, інакше всі проєкти, які створюються на блокчейні Ethereum, існували б у їхніх власних сховищах. З таким принципом роботи було б дуже важко здійснювати перекази будь-якого виду.

Якщо говорити простою мовою, стандарт ERC20 визначає набір функцій [7], які повинні виконувати всі токени ERC20 для забезпечення інтеграції з іншими контрактами, гаманцями або ринками. Цей набір функцій досить короткий і простий (див. рис. 1.4).

```
function totalSupply() public view returns (uint256);
function balanceOf(address tokenOwner) public view returns (uint);
function allowance(address tokenOwner, address spender) public view returns (uint);
function transfer(address to, uint tokens) public returns (bool);
function approve(address spender, uint tokens) public returns (bool);
function transferFrom(address from, address to, uint tokens) public returns (bool);
```

Рисунок 1.4 – Стандартні функції ERC20

Функції ERC20 дають змогу зовнішньому користувачеві, скажімо, застосунку крипто-гаманця, дізнатися баланс користувача та переказати кошти від одного користувача до іншого з належною авторизацією.

Також у розумному контракті визначено дві конкретні події (Підтвердження і Переказ) (див. рис. 1.5).

```
event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
event Transfer(address indexed from, address indexed to, uint tokens);
```

Рисунок 1.5 – Стандартні подій ERC20

Ці події будуть викликані або опубліковані, коли користувачеві буде надано права на отримання токенів з рахунку і після того, як токени будуть за

фактом переведені.

Після короткого розгляду стандарту класичного взаємозамінного токена в мережі Ethereum ми зможемо краще розуміти, як влаштовані контракти для взаємодії з ними.

1.5 Порівняння сучасних мов програмування для написання смарт-контрактів

Якщо ми говоримо про мови, які підтримують смарт-контракти, то ми можемо виділити приблизно такий список: Move, Bitcoin Script, Solidity, Vyper, Go, C/C++, C#, Rust, Haskell, Clarity. Створення окремої мови програмування під свій власний блокчейн стало поширеною практикою в цій індустрії. Так, наприклад, були створені такі мови як Solidity, Vyper, Simplicity або Michelson.

Ось основні нюанси, які я враховував під час вибору мови [8]:

– які блокчейни використовують цю мову як мову смарт-контрактів;
– синтаксис якої традиційної мови програмування найбільше схожий на цю мову;
– чи є ця мова повною за Тюрінгом;
– чи підтримує ця мова цикли;
– це інтерпретована мова чи компільована мова.

Якщо враховувати вище перераховані фактори і той факт, що блокчейн, з яким я б хотів взаємодіяти, є Ethereum, мій список мов звузився до Vyper, Serpent, LLL та Solidity. Оскільки Solidity є основною і найпоширенішою мовою для написання смарт-контрактів в Ethereum, а також є багато легкодоступного матеріалу з його вивчення мій вибір припав саме на нього.

1.6 Огляд і вибір інструментів розробки та обгрантування вибору Hardhat

Hardhat – це середовище розробки Ethereum для розробників, яке дозволяє виконувати часті завдання [9].

Крім простого розгортання контрактів і запуску тестів, Hardhat надає ще кілька функцій, які роблять його потужнішим і унікальнішим.

Hardhat постачається в комплекті з Hardhat Network – локальною нодою мережі Ethereum для розроблення, що надає безліч функцій, як-от автоматичні повідомлення про помилки, форкінг мережі та режими майнінгу. Hardhat вже за замовчуванням використовуєте Hardhat Network. Вона видобуває блок з кожною отриманою транзакцією, по порядку і без затримок.

У розробці Hardhat використовується 30+ плагінів, тому вони вважаються основою Hardhat.

Chai. Бібліотека Chai – це набір інструментів для тестування коду на JavaScript [10]. Вона дозволяє розробникам створювати тести, які перевіряють правильність роботи функцій та методів у вашому JavaScript-коді. Chai надає декілька різних способів для написання тестів, включаючи BDD- та TDD-стилі. Бібліотека також дозволяє використовувати різні методи перевірки, такі як assert, expect та should. Використання chai допомагає забезпечити якість коду та зменшити кількість помилок, що можуть виникнути під час розробки.

Traffle. Truffle – це середовище розробки [11], що належить до блокчейну Ethereum. Truffle дуже популярний фреймворк і за ним стоїть велике ком'юніті розробників. Truffle Suite – це екосистема Web 3 розробки, що складається з 3х різних інструментів: Truffle, Ganache, і Drizzle. Truffle допомагає керувати артефактами смарт-контрактів, щоб ви могли зосередитися на інших частинах процесу розробки та витрачати менше часу на організацію файлів.

1.7 Технічне завдання

1.7.1 Опис предметної області

Розробляється смарт-контракт для створення чесного голосування. Користувач буде мати можливість взяти участь у голосуванні у ролі кандитата якщо власник додав його в голосування. Будь який інший адрес матиме змогу проголосувати за обраного кандидата у період голосування. Переможець повинен мати можливість отримати свої кошти після завершення голосування. Тільки власник може додавати учасників або голосування, також тільки власник може редагувати параметри голосування. Після отримання виграшу переможцем, тільки власник має можливість на отримання комісії з проведеного голосування.

1.7.2 Функціональні вимоги

Функціональне призначення смарт-контракту – реалізувати можливість створювати, брати участь та редагувати голосування.

Загальні функціональні можливості смарт-контракту:

а) для власника:

- взяти участь у голосуванні;
- переглядання інформації про голосування;
- переглядання наявності кандидата у списку;
- додавання голосування;
- виведення призу;
- зміна періоду голосування;
- додавання кандидата до голосування;
- видалення кандидата;
- задавання максимальної кількості кандидатів;

б) для кандидата:
   - взяти участь у голосуванні;
   - проголосувати за обраного кандидата;
   - переглядання інформації про голосування;
   - переглядання наявності кандидата у списку;
   - виведення призу;

в) для всіх інших адрес:
   - проголосувати за обраного кандидата;
   - переглядання інформації про голосування;
   - переглядання наявності кандидата у списку.

1.7.3 Нефункціональні вимоги

Безпека: надійність способу голосування та складність злому особистої адреси, надійність зберігання і неможливість впливу на кінцевий результат шляхом злому системи за рахунок способу зберігання даних у ній.

Надійність: неможливість відключення голосування, оскільки доступ до нього не відбувається через конкретний сервер.

Виконання стандартів: написання смарт-контракту відповідно до стандартів захисту від вразливостей та економії газу.

1.7.4 Опис основних етапів створення смат-контрактів

Перший етап розробки смарт-контрпакту – це вибір на якій блокчейн-платформі його створити. Ethereum, як і раніше, зберігає лідерство за кількістю запущених смарт-контрактів і Dapps [12], але він повільний, дорогий і має великі проблеми з масштабуванням, тому все більше стартапів обирають інші блокчейни: Polygon, Polkadot, Cardano, Solana, BSC, Tezos або

Hyperledger.

Другим етапом створення смарт-контракту можна назвати вибір інструментів розробки. Як і у випадку з блокчейнами, існує також кілька інструментів, які розробник може використовувати для створення смарт-контракту. Про деякі з них я вже встиг розповісти вище.

На третьому етапі відбувається проєктування та запис коду смарт-контракту. У кожного блокчейна є свій набір інструментів для розробки. Наприклад, OpenZeppelin часто використовується для смарт-контрактів на Ethereum [13].

Четвертим етапом ми безумовно можемо назвати тестування смарт-контракту. З огляду на ту кількість зломів смарт-контрактів, яку ми спостерігаємо, тестування, ймовірно, є найважливішим етапом розробки смарт-контрактів. Річ у тім, що смарт-контракти – це програмне забезпечення з відкритим вихідним кодом, а отже, будь-який хакер може вивчити його код і знайти "дірки", щоб використати їх для злому смарт-контракту. Крім того, після того як ви створите свій смарт-контракт і розгорнете його, змінити його буде неможливо, тож тестування – це останній шанс для розробника усунути можливі недоліки і помилки. Для спрощення тестування у блокчейнів зазвичай є тестові мережі, що дають змогу провести перевірку смарт-контракту без будь-яких ризиків втрати грошей, даних і репутації. Вибір тестової мережі залежить від блокчейна.

Останній етап розробки смарт-контракту – це його розгортання в середовищі блокчейну. Після цього смарт-контракт стане доступним користувачам і його не можна буде ніяк змінити.

1.8 Висновки до розділу 1

В першому розділі було розглянуто нову у використанні технологію блокчейн та способи її застосування. Переглянуто сучасні мови

програмування які підходять для написання смарт-контрактів у різних блокчейнах. Порівняно та обрано інструменти для розробки системи голосування, а також описані основні на мою думку єтапи розробки смарт-контрактів.

2 ПРОЄКТУВАННЯ

2.1 Загальне призначення

Основним призначенням розробки системи голосування було надання змоги використання технології блокчейн для зручного та чесного голосування. Основна мета – полегшення процесу слідкування та рішення проблеми довіри у вирішенні подібних завдань.

2.2 Створення діаграм

Діаграми – це візуальне представлення даних та інформації, що дає змогу простіше та наочніше її сприймати й аналізувати дані або, якщо брати поточну роботу, схему роботи контракту.

2.2.1 Діаграма прецедентів

Діаграма прецедентів – це графічне зображення, що описує взаємодію між користувачем та системою. Вона використовується в процесі аналізу та проєктування програмного забезпечення для визначення функціональних вимог до системи та її компонентів.

Діаграма прецедентів складається з акторів та прецедентів. Актор – це користувач системи або зовнішній компонент, який взаємодіє з системою. Прецедент – це функціональна діяльність, яку система виконує для задоволення потреб користувачів або інших акторів. Діаграма прецедентів дозволяє зрозуміти, як система взаємодіє з користувачем (див. рис. 2.1).

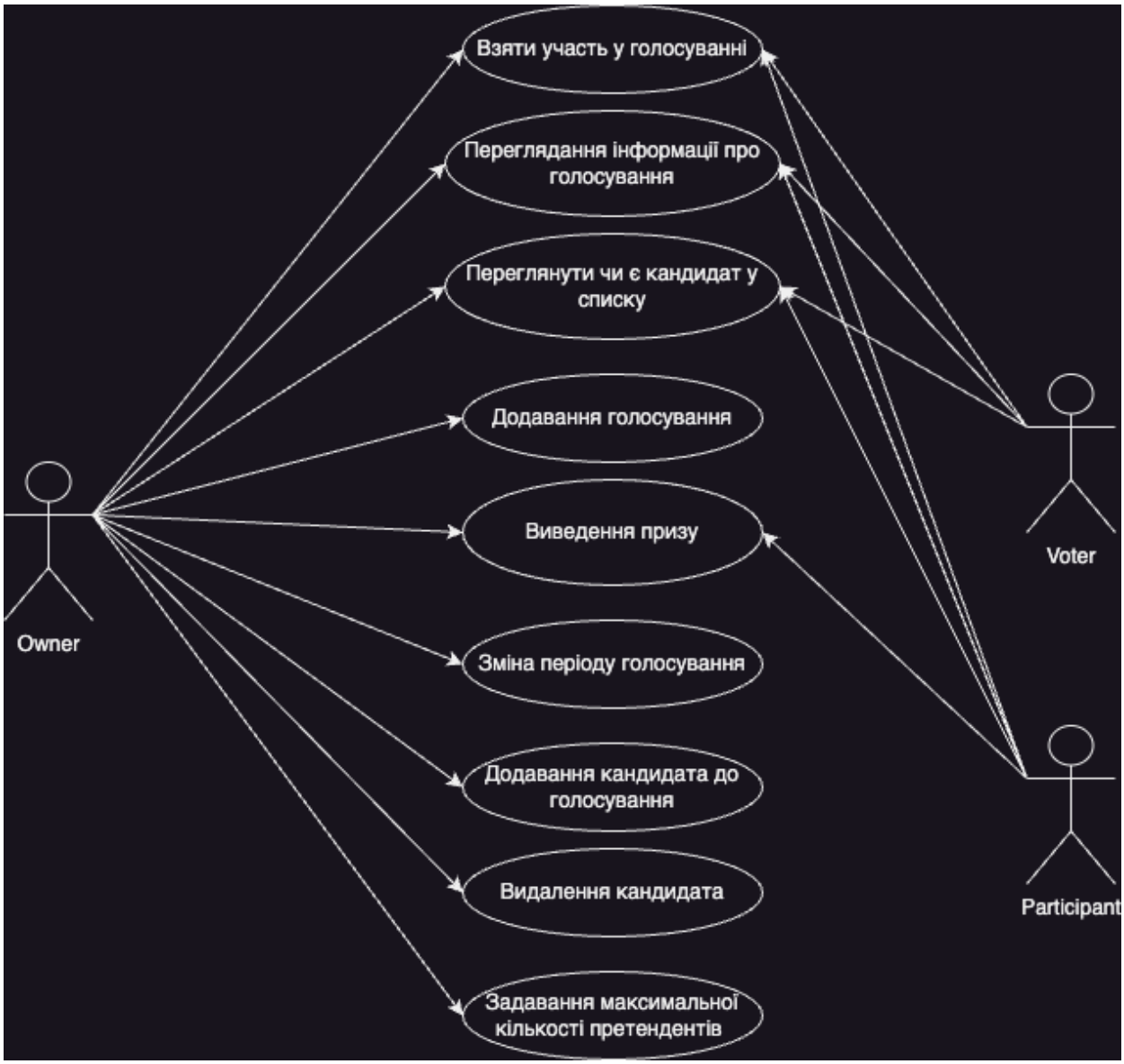

Рисунок 2.1 – Діаграма прецедентів контракту системи голосування

Наведемо опис прецедентів.

Прецедент «Взяти участь у голосуванні» – надає можливість “користувачу” внести якусь кількість ETH за обраного учасника.

Прецедент «Переглядання інформації про голосування» – надає можливість кожному охочому переглянути інформацію про голосування.

Прецедент «Переглянути чи є кандидат у списку» – надає можливість кожному охочому переглянути чи є n “кандидат” у n голосуванні.

Прецедент «Додавання голосування» – надає можливість “власнику” контракта додавати голосування у чорнетку.

Прецедент «Виведення призу» – надає можливість “переможцю” запросити власну нагороду після закінчення голосування, а також вишукує та відправляє комісію “власнику”.

Прецедент «Зміна періоду голосування» – надає можливість “власнику” змінювати період голосування доки він знаходеться у чорнетці.

Прецедент «Додавання кандидата до голосування» – надає можливість “власнику” додавати голосування доки воно не разпочалось.

Прецедент «Видалення кандидата» – надає можливість “власнику” видаляти “кандидата” доки голосування не почалося.

Прецедент «Задавання максимальної кількості кандидатів» – надає можливість “власнику” задавати максимальну кількість “кандидатів” у голосуванні.

2.2.2 Функціональна діаграма

Функціональна діаграма є графічним зображенням функцій та процесів, які виконує система або програма. Вона дозволяє описати функціональні можливості системи та взаємодію між її складовими частинами. Функціональна діаграма допомагає розробникам та користувачам системи зрозуміти, як вона працює та які функції вона виконує. Вона дозволяє виявити можливі проблеми та покращити ефективність системи. У випадку смарт-контракту, функціональна діаграма допомагає визначити, які функції він повинен виконувати, які дані має обробляти та які відповіді має здобувати (див. рис. 2.2).

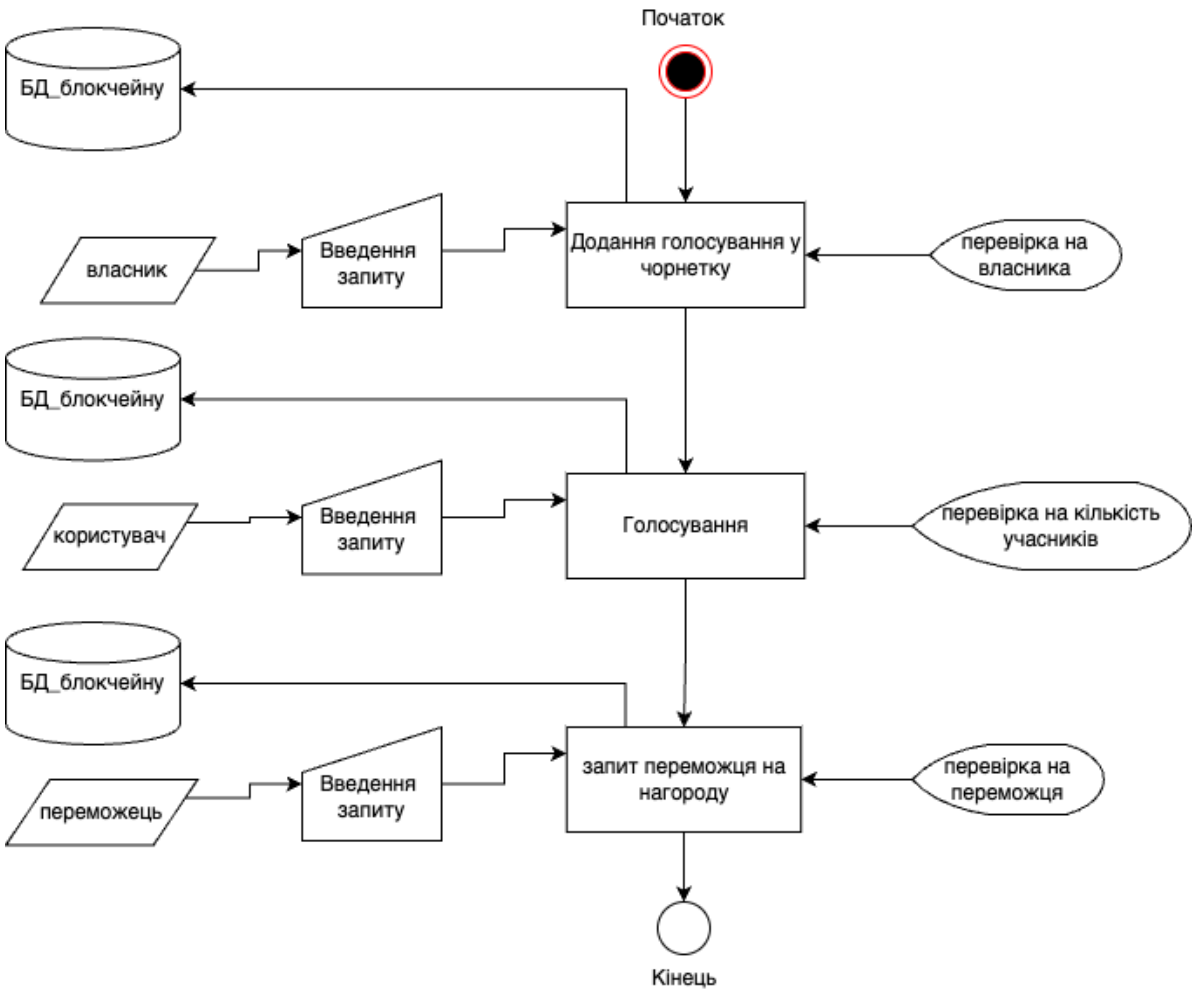

Рисунок 2.2 – Функціональна діаграма живого циклу голосування

2.2.3 Структурна схема

Структурна схема – це графічний засіб, який використовується для відображення структури системи та взаємозв'язків між її компонентами. Вона дає змогу візуально представити складну систему в простій формі, що допомагає розробникам та аналітикам зрозуміти її структуру та логіку роботи. Структурна схема може містити блоки, які відображають окремі компоненти системи, такі як модулі, класи, функції або об'єкти. Кожен блок може мати вхідні та вихідні порти, які відображають взаємодію з іншими блоками. Структурна схема може бути використана для проєктування, чи візуалізації взаємодії компонентів (див. рис. 2.3).

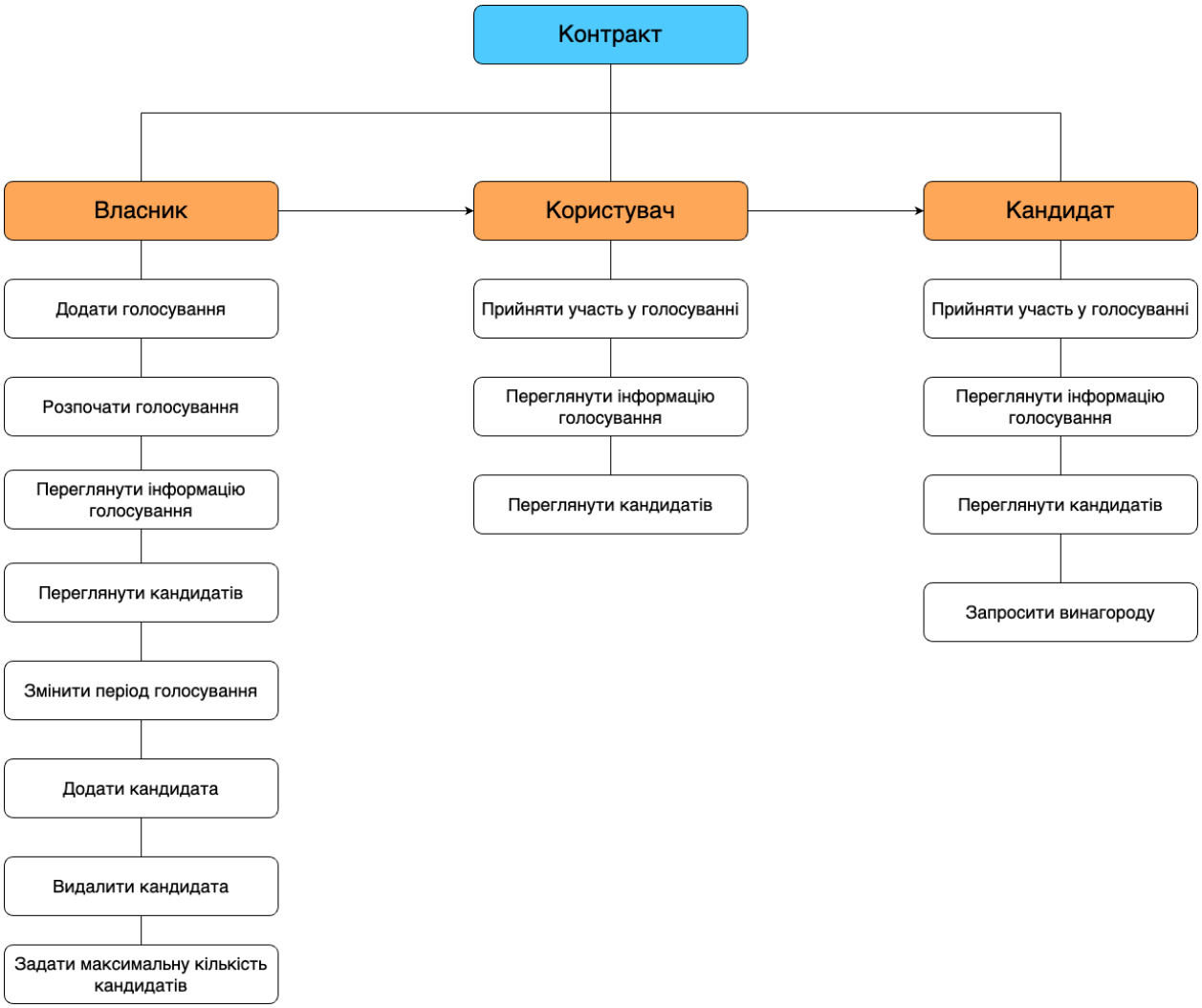

Рисунок 2.3 – Функціональна діаграма живого циклу голосування

2.3 Аналіз паттернів проєктування у Solidity

Якщо ми говоримо про смарт-контракти, знання патернів – це один з найважливіших моментів, оскільки найчастіше ми працюємо безпосередньо з активами інших людей і найменші помилки в логіці контракту можуть вплинути на виникнення слабкостей у його захисті та подальшим зняттям коштів зловмисниками. Саме з цієї причини смарт-контракти проходять незалежні перевірки і тому основні з патернів я опишу нижче.

Патерн Timelock. Timelock це патерн, який дає змогу поставити транзакцію в чергу і відкласти на якийсь період [14]. Timelock може

використовуватися, наприклад, для розгляду будь-яких дій, запропонованих у транзакції та скасування транзакції через непотрібність до її початку.

Патерн MultiSig. MultiSig це патерн який який реалізує можливість виконувати будь-які дії тільки після досягнення якоїсь кількості голосів. Припустимо, у нас є завдання захистити від несанкціонованого доступу кошти на адресі, де зберігаються гроші організації. Доступ до адреси є в кількох людей. Щоб захистити гроші, якщо когось із цих людей силою змусить підписати виведення, існує multisig [15].

Тепер кожен із трьох учасників, бажаючи створити транзакцію на виведення, повинен надати свій підпис, що підтверджує його згоду з транзакцією. Коли набирається достатньо підписів, кошти переводяться. Така логіка і носить назву multisig: надсилання N з M підписів (M > N) для підтвердження операції.

Для децентралізованих мереж multisig – рідний патерн, тому що будь-яка валідна транзакція, надіслана на деяку адресу, by design містить у собі електронний підпис, створений секретним ключем відправника, тож практично вся необхідна для multisig-адрес функціональність уже на борту більшості сучасних блокчейнів.

Найцікавішою є найпростіша multisig-адреса 2/3, виведення з якої можливе тільки за згодою двох із трьох учасників. Такий multisig здатний зробити зручними і безпечними багато угод із трьома сторонами і вирішувати завдання, коли двоє учасників довіряють третьому розсудити їх. Практично будь-які фінансові послуги, в яких є третя довірена сторона, реалізуються за допомогою multisig 2/3.

Патерн Commit/Reveal. Патерн commit/reveal створений для унеможливлення розкриття результату до того моменту, коли це буде обумовлено логікою смарт-контракту

Уявімо "гру камінь, ножиці, папір". Оскільки в середовищі блокчейну неможливо одночасно розкрити вибір, користувачі мають заздалегідь зафіксувати свій вибір. Користувач 1 обирає "папір". Він хешує його разом із

випадковим числом (званим сіллю) і відправляє хеш у смарт-контракт. Користувач 2 робить те саме зі своїм вибором "камінь". Таким чином, вони підтвердили свій вибір, не розкриваючи його.

Тепер користувач 1 розкриває свій вибір, публікуючи "папір" і сіль. Потім контракт може перевірити, чи відповідає він зафіксованому хешу. Якщо це відповідає, це дійсно подання. Те саме стосується і користувача 2.

Якщо будь-який із користувачів спробує змінити свій вибір і замість цього покаже щось інше, більш корисне, це буде виявлено шляхом порівняння зафіксованого хешу з хешем із виявлених значень [16].

Схема розкриття фіксації може бути доповнена механізмом покарання, якщо користувач не розкриває її істинне значення або не розкриває що-небудь у заданий час (див. рис. 2.4).

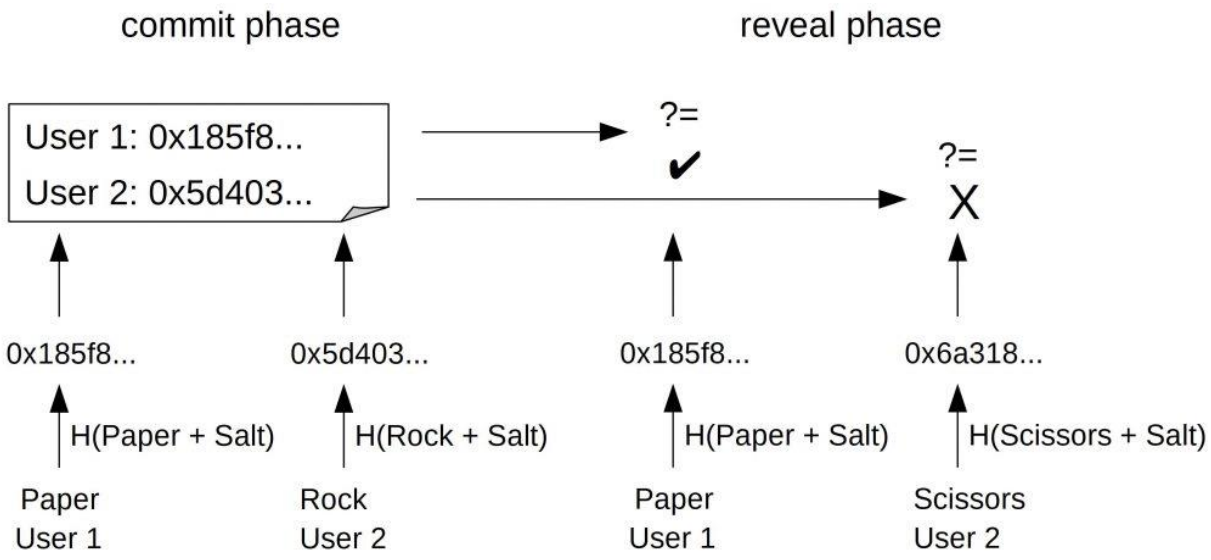

Рисунок 2.4 – Приклад схеми патерну Commit/Reveal/

Патерн Pull over Push. Надсилання ефіру на іншу адресу в Ethereum передбачає виклик одержувача. Існує кілька причин, через які цей зовнішній виклик може завершитися помилкою. Якщо адреса одержувача є контрактом, у ній може бути реалізована резервна функція, яка просто видає виняток після її виклику. Ще одна причина поломки – скінчився бензин. Це може статися в тих випадках, коли необхідно виконати безліч зовнішніх викликів у

межах одного виклику функції, наприклад, під час надсилання прибутку від ставки кільком переможцям. З цих причин розробники повинні дотримуватися простого принципу: ніколи не довіряйте зовнішнім викликам виконання без видачі помилки. У більшості випадків це не проблема, тому що можна стверджувати, що одержувач зобов'язаний переконатися, що він може отримати свої гроші, а в разі, якщо він цього не зробить, це йому тільки на шкоду.

Патерн Pull over Push складається з трьох учасників. По-перше, юридична особа, відповідальна за ініціювання передачі (наприклад, власник контракту або сам контракт), починає процес. По-друге, смарт-контракт відповідає за відстеження всіх балансів. Третій учасник – це одержувач, який не просто отримає свої кошти через транзакцію, але має активно запитувати виведення коштів, щоб ізолювати процес від іншої логіки виплат і контрактів [17].

Щоб ізолювати всі зовнішні виклики один від одного та логіки контракту, патерн Pull over Push перекладає ризик, пов'язаний із переказом ефіру, на користувача, дозволяючи йому вивести (pull) певну суму, яку інакше довелося б надіслати до нього (штовхати). Основним компонентом цієї реалізації є зіставлення, яке відстежує непогашені залишки користувачів. Замість фактичного переказу ефіру від контракту до одержувача викликається функція, яка додає до відображення запис про те, що користувач має право зняти зазначену суму. У разі, якщо в зіставленні вже є запис для цієї адреси, сума додається до наявної. Тепер користувач несе відповідальність за зняття коштів, виконавши транзакцію за допомогою методу виведення смарт-контракту, що використовує шаблон Checks Effects Interactions для оновлення непогашеного балансу перед фактичним переданням ефіру.

Реалізоване таким чином, викинуте виключення в одній із передач впливатиме тільки на цю конкретну передачу, а не на цілу серію передач або навіть на весь контракт.

2.4 Проєктування з урахуванням захисту від різних типів атак

Як я вже встиг описати вище, розробниг solidity не завжди має використовувати усі паттерни, але він мае знати їх та деякі види аттах що можуть зустрічатися на просторах блокчейну. Таким чином я опишу кілька з них нижче і покажу як захиститися від деяких з них.

2.4.1 Dos атака

Відмова в обслуговуванні (звідси іменований DoS) забороняє законним користувачам використовувати смарт-контракти постійно або протягом певного періоду часу [18].

Також ми можемо розглянути в даному прикладі підхід Push/Pull, представлений у патернах раніше (див. рис. 2.5).

```
// push
    function refund() external {
        for(uint i = refundProgress; i < allBidders.length; i++) {
            address bidder = allBidders[i];
            (bool success,) = bidder.call{value: bidders[bidder]}("");
            require(success, "failed!");
            refundProgress++;
        }
    }
```

Рисунок 2.5 – Приклад вразливого коду під Dos атаку

Атака в даному випадку реалізується з боку одного з одержувачів таким чином, що не дає можливість відправити кошти на свою адресу і тим самим зупиняє відправлення коштів усім наступним одержувачам (див. рис. 2.6).

```
receive() external payable {
        if(hack == true) {
            while(true) {}
        } else {
            owner.transfer(msg.value);
        }
```

Рисунок 2.6 – Приклад реалізації Dos атаки

Найшвидший спосіб обходу такої нехитрої атаки це використовувати pull замість push. Інакше кажучи, ми не видаємо кошти одразу всім, ми створюємо функцію, через яку вже користувачі контракту можуть звертатися для перевірки та виведення своїх коштів, звісно, з підготовленим захистом від reentrancy, про який ви дізнаєтеся в наступному пункті плану. Також ми запросто можемо підготувати обхід адрес виплати на які не змогли прийти, додаючи їх в окремий масив для подальшого розбору (див. рис. 2.7).

```
function refund() external {
        for(uint i = refundProgress; i < allBidders.length; i++) {
            address bidder = allBidders[i];
            (bool success,) = bidder.call{value: bidders[bidder]}("");
            if(!success) {
                  //failedRefunds.push(bidder);
            }
            refundProgress++;
        }
    }
```

Рисунок 2.7 – Приклад рішення Dos вразливості

2.4.2 Reentrancy атака

Атака Reentrancy – одна з найбільш руйнівних атак у смарт-контракті Solidity. Атака з повторним входом відбувається, коли функція виконує зовнішній виклик іншого ненадійного контракту [19].

Уявіть, що є контракт, у якому міститься 1000 ETH, і у сотні

користувачів є можливість зняти з цього контракту по 10 ETH, а тепер поглянемо на типову функцію виведення коштів такого контракту (див. рис. 2.8).

```
function refund() external noReentrancy {
        uint refundAmount = bidders[msg.sender];
        if (refundAmount > 0) {
           (bool success,) = msg.sender.call{value: refundAmount}("");
            require(success, "failed!");
            bidders[msg.sender] = 0; //ось і помилка
        }
    }
```

Рисунок 2.8 – Приклад Reentrancy вразливості

Одже ми бачимо що спочатку ми запитуємо наші кошти, а вже потім прирівнюємо їх до нуля для неможливості зняти їх надалі. Тут і криється груба помилка, оскільки в момент виклику коштів ми можемо використовувати функцію receive() вже на нашому контракті для повторного виклику функції refund() (див. рис. 2.9).

```
receive() external payable {
        if(auction.currentBalance() >= 10) {
            auction.refund();
        }
    }
```

Рисунок 2.9 – Приклад коду для створення Reentrancy атаки

Таким чином ми можемо рекурсивно знімати кошти з контракту із загальними коштами поки вони не закінчаться.

Найпростіший і дієвий спосіб захисту від такої атаки буде обнулити рахунок одержувач на загальному контракті до виклику зняття (див. рис. 2.10). Пізніше я використаю цей спосіб захисту у своєму контракті.

```
function refund() external noReentrancy {
      uint refundAmount = bidders[msg.sender];
      if (refundAmount > 0) {
         bidders[msg.sender] = 0;
         (bool success,) = msg.sender.call{value: refundAmount}("");
         require(success, "failed!");
      }
   }
```

Рисунок 2.10 – Приклад коду для захисту Reentrancy атаки

2.5 Висновки до розділу 2

У другому розділі ми проаналізували необхідні для розуміння патерни проєктування Solidity, ознайомилися з часто використовуваними вразливостями під час проєктування і розібрали приклади для їхнього розв'язання, також ми склали кілька UML діаграм для кращого розуміння структури контракту.

3 СТВОРЕННЯ ТА РОЗГОРТАННЯ СМАРТ КОНТРАКТУ

3.1 Встановлення середовища розробки Remix та додаткових бібліотек

Як уже було описано , під час першого і другого етапу розроблення ми використовуватимемо як IDE Remix (див. рис. 3.1) для написання контракту та зручного власного тестування завдяки локальнуму середовищу яке надає Remix. Також пізніше будут використані Visual Studio Code для зручного покриття контракту тестами [20].

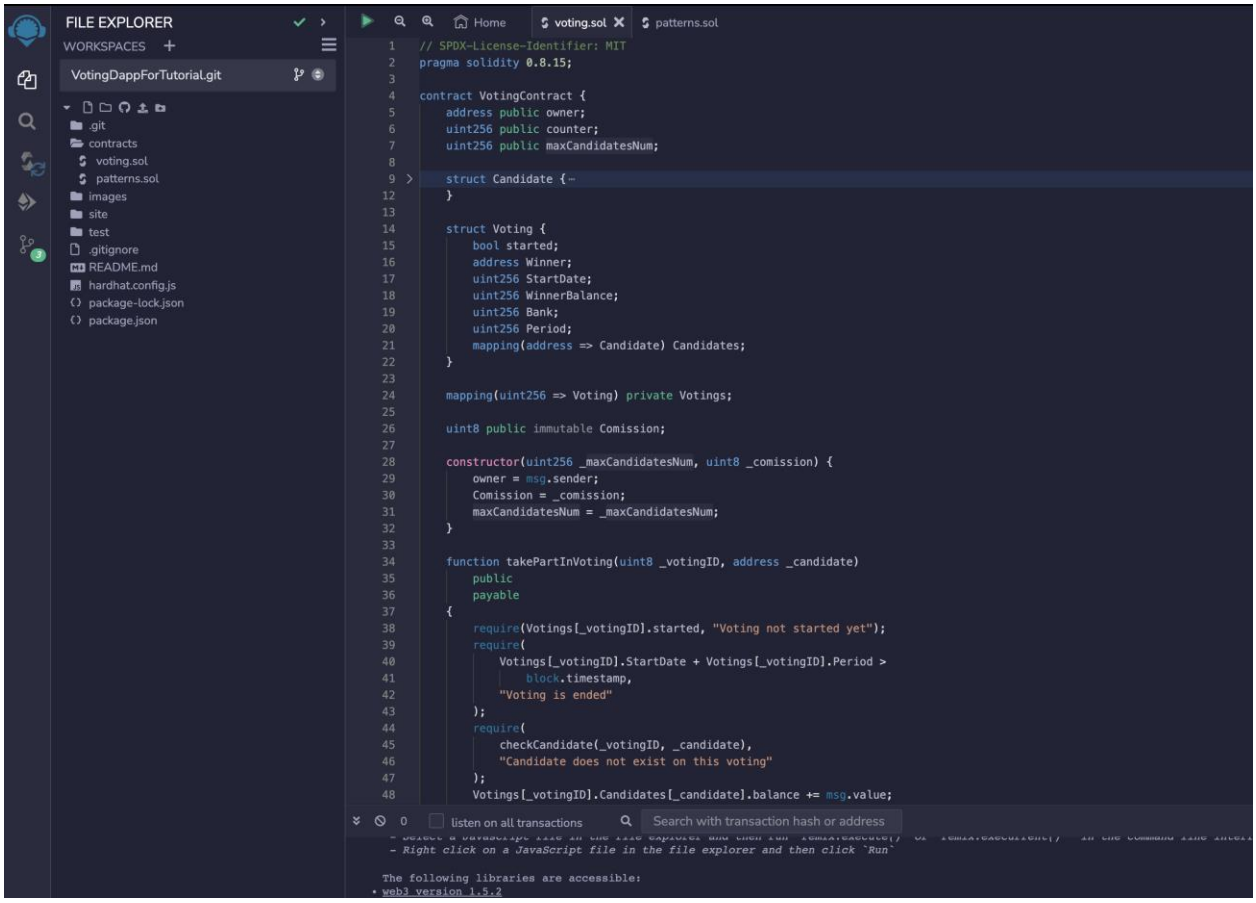

Рисунок 3.1 – Редактор Remix IDE

За період використання Remix IDE найкориснішими виявилися функції взаємодії з Github вбудовані в Remix. А також деплой контракту в локальне

середовище і підсвічування помилок під час компіляції, це дало змогу прискорити цикл написання, компіляція, деплой стосовно інших редакторів коду.

3.2 Написання смарт-контракту

На першому етапі написання контракту ми оголошуємо змінні та структури даних, які використовуються далі. Однією з особливостей написання контрактів, яку ми повинні брати до уваги, є економія пам'яті. Це є доволі важливим моментом під час проєктування, бо деплой контракту в основну мережу завжди коштуватиме вам будь-якої суми монет, що використовуються в якості оплати тим чи іншим блокчейном. Найпростішим і використовуваним кожним розробником способом економії газу є запис змінних одного типу поруч, бо дані зберігаються блоками по 32 байти [21] і мають можливість об'єднуватися в разі, якщо це дані одного типу і розміру та їх об'єднання в один блок пам'яті не перевищує 32 байти (див. рис. 3.2).

```
address public owner;
    uint256 public counter;
    uint256 public maxCandidatesNum;

    struct Candidate {
        uint256 balance;
        bool isExistOnThisVoting; //перевірка на наявність у списку голосуванні
    }

    struct Voting {
        bool started;
        address Winner;
        uint256 StartDate;
        uint256 WinnerBalance;
        uint256 Bank;
        uint256 Period;
        mapping(address => Candidate) Candidates;
    }

    mapping(uint256 => Voting) private Votings;

    uint8 public immutable Comission;
```

Рисунок 3.2 – Групування даних для економії газу

На наступному кроці ми створюємо конструктор з встановленням власника контракту, комісії за голосування та максимальну кількість голосування (див. рис. 3.3).

```
constructor(uint256 _maxCandidatesNum, uint8 _comission) {
        owner = msg.sender;
        Comission = _comission;
        maxCandidatesNum = _maxCandidatesNum;
    }
```

Рисунок 3.3 – Кокнструктор контракту

Пізніше ми встановимо перевірку щоб виконання деяких функцій було доступно лише власнику контракта. Не рідко при деплої контракту ми всановлюємо декілька адрес як власників або реалізуємо паттерн мультипідпису для зручності використання контракту.

Другим кроком буде реалізування функції прийняття участі у голосуванні. Функції які можуть отримувати чи віддавати токени ми помічаємо як payable. Функція реалізована по принципу царя гори, щоб кандидат зайняв роль переможця, недостатньо надіслати в банк кандидата суму токенів щоб банки кандидата і поточного переможця були рівні, необхідно збільшити банк банк кандидата хоча б на 1 Gwei для зміни поточного лідера (див. рис. 3.4).

```
function takePartInVoting(uint8 _votingID, address _candidate)
        public
        payable
    {
        require(Votings[_votingID].started, "Voting not started yet");
        require(
            Votings[_votingID].StartDate + Votings[_votingID].Period >
                block.timestamp,
            "Voting is ended"
        );
        require(
            checkCandidate(_votingID, _candidate),
            "Candidate does not exist on this voting"
```

```
        );
        Votings[_votingID].Candidates[_candidate].balance += msg.value;
        Votings[_votingID].Bank += msg.value;
        //перевіряємо чи не змінився лідер після кожного нового голосу
        if (
            Votings[_votingID].Candidates[_candidate].balance >
            Votings[_votingID].WinnerBalance
        ) {
            Votings[_votingID].WinnerBalance =
Votings[_votingID].Candidates[_candidate].balance;
            Votings[_votingID].Winner = _candidate;
        }
    }
```

Рисунок 3.4 – Реалізація функції голосування у контракті

Як ми вже знаємо, автоматичний вивід може викликати деякі труднощі в коді, і за певної реалізації дає змогу реалізувати Dos атаку стороні, що викликає. Для обходу цього типу атак можливість виведення коштів реалізовано як функцію, яку має викликати кандидат із найбільшою кількістю коштів у банку для отримання нагороди (див. рис. 3.5).

```
function WithdrowMyPrize(uint256 _votingID) public {
        require(Votings[_votingID].started, "Voting not started yet");
        require(
            Votings[_votingID].StartDate + Votings[_votingID].Period <
                block.timestamp,
            "Voting is not over yet!"
        );
        require(
            msg.sender == Votings[_votingID].Winner,
            "You are not a winner!"
        );
        require(
            Votings[_votingID].Bank > 0,
            "You have already received your prize!"
        );
        uint256 amount = Votings[_votingID].Bank;
        uint256 ownersComission = (Comission * amount) / 100; //вишукуємо нашу комісію
        uint256 clearAmount = amount - ownersComission;
        Votings[_votingID].Bank = 0; //захист від reentrancy attack
        payable(owner).transfer(ownersComission);
        payable(msg.sender).transfer(clearAmount);
    }
```

Рисунок 3.5 – Реалізація функції виведення і захист від reentrancy attack

Усі функції, які впливають на стан блокчейна, потребують плати, оскільки кожна зміна вимагає обчислювальної потужності валідаторів. Але маючи власний вузол або підключаючись до сервісів, які надають такі послуги, ми можемо без витрат на газ переглядати різні змінні та структури контрактів. Таким чином було реалізовано функцію перегляду інформації про n голосування за його ID (див. рис. 3.6).

```
function getVotingInfo(uint256 _votingID)
        public
        view
        returns (
            bool,
            uint256,
            uint256,
            uint256,
            uint256,
            address
        )
    {
        return (
            Votings[_votingID].started,
            Votings[_votingID].StartDate,
            Votings[_votingID].Period,
            Votings[_votingID].WinnerBalance,
            Votings[_votingID].Bank,
            Votings[_votingID].Winner
        );
    }
```

Рисунок 3.6 – Реалізація функції перегляду інформації про голосування

Так само було реалізовано функцію перегляду чи є n адреса кандидатом m голосування (див. рис. 3.7).

```
function checkCandidate(uint256 _votingID, address _candidate)
        public
        view
        returns (bool)
    {
        return (Votings[_votingID].Candidates[_candidate].isExistOnThisVoting);
    }
```

Рисунок 3.7 – Реалізація функції перегляду адреси в голосуванні

Далі ми розглядатимемо функції, доступні лише користувачеві, який розгорнув цей контракт, оскільки саме він у конструкторі встановлюється як owner. Під час додавання голосування ми створюємо emit для зручної взаємодії контракту з інтерфейсом у майбутньому.

Emit в Solidity – це ключове слово, яке використовується для генерації подій (events) в контрактах Ethereum. Події є способом повідомлення про відбування певних дій у контракті, які можуть бути перехоплені та оброблені ззовні. Ключове слово emit використовується в контексті виклику функції, яка спричиняє подію. Наприклад, якщо у контракті є функція, яка змінює стан, можна використовувати emit для генерації подій.

У всіх важливих для перегляду на мою думку функціях я використовую emit для створення подій, пізніше це можна використати для зручного відображення даних у різних DApps (див. рис. 3.8).

```
function addVoting(uint256 _period, address[] calldata _candidates) // созд. голосование с
периодом
    public
    onlyOwner
  {
    require(_candidates.length < maxCandidatesNum, "Too many candidates!");
    Votings[counter].Period = _period;
    for (uint256 i = 0; i < _candidates.length; i++) {
      addCandidate(counter, _candidates[i]);
    }
    emit votingDraftCreated(counter);
    counter++;
  }
```

Рисунок 3.8 – Реалізація функції створення голосування та події

Для більшості замовників неможливість змінювати дані в блокчейні нерідко виливається в проблему, клієнт часто не може користуватися комфортно тим, що не може змінити. Саме тому при створенні голосування воно не почнеться доти, доки його не підтвердить власник (див. рис. 3.9).

```
function startVoting(uint256 _votingID) public onlyOwner {
        Votings[_votingID].started = true;
        Votings[_votingID].StartDate = block.timestamp;
        emit votingStarted(_votingID, block.timestamp);
    }
```

Рисунок 3.9 – Реалізація функції старту яка також відіграє роль чернетки

Редагувати час періоду голосування, додавання, видалення або зміна максимальної кількості кандидатів – це також функції, які можуть бути доступні тільки власнику контракту (див. рис. 3.10).

```
function editVotingPeriod(uint256 _votingID, uint256 _newPeriod) //смена периода
        public
        onlyOwner
    {
        require(
            Votings[_votingID].started == false,
            "Voting has already begun!"
        );
        Votings[_votingID].Period = _newPeriod;
    }
    function addCandidate(uint256 _votingID, address _candidate)
        public
        onlyOwner
    {
        require(
            Votings[_votingID].started == false,
            "Voting has already begun!"
        );
        Votings[_votingID].Candidates[_candidate].isExistOnThisVoting = true;
        emit candidateInfo(_votingID, _candidate, true);
    }
    function deleteCandidate(uint256 _votingID, address _candidate)
        public
        onlyOwner
    {
        require(
            Votings[_votingID].started == false,
            "Voting has already begun!"
        );
        Votings[_votingID].Candidates[_candidate].isExistOnThisVoting = false;
        emit candidateInfo(_votingID, _candidate, false);
    }
    function setMaxCandidatesNum(uint256 _maxCandidatesNum) public onlyOwner {
        maxCandidatesNum = _maxCandidatesNum;
    }
```

Рисунок 3.10 – Реалізація функцій редагування голосування

Для кожної функції доступу до якої не передбачається нікому крім власника контракту ми встановлюємо модифікатор (див. рис. 3.11).

```
modifier onlyOwner() {
        require(
            msg.sender == owner,
            "Error! You're not the smart contract owner!"
        );
        _;
    }
```

Рисунок 3.11 – Реалізація класичного модифікатора Owner

Далі ми прописуємо івенти з індексованими полями для зручної взаємодії з фронт-ендом і після етапу розробки коду контракту переходимо до етапу тестування.

3.3 Встановлення Hardhat і залежностей

На цьому етапі я переніс свій проєкт у Visual Studio Code для зручної взаємодії з hardhad. Потім була встановленна платформа Node.js диспетчер версій узлів NVM.

NVM є інструментом для управління версіями Node.js на вашому комп'ютері. Він дозволяє швидко та легко встановлювати, видаляти та перемикатися між різними версіями Node.js. Також було встановлено систему контролю версій git.

Після підготування ми можемо створити та перейти у папку де розгорнемо на hardhad проєкт (див. рис. 3.12).

Також HardHat пропонує встановити нам toolbox з різними корисними бібліотеками, як-от Mocca, Chai та іншими (див. рис. 3.13), пізніше вони знадобляться нам для написання тестів.

На цьому підготовка середовища для написання тестів була завершена.

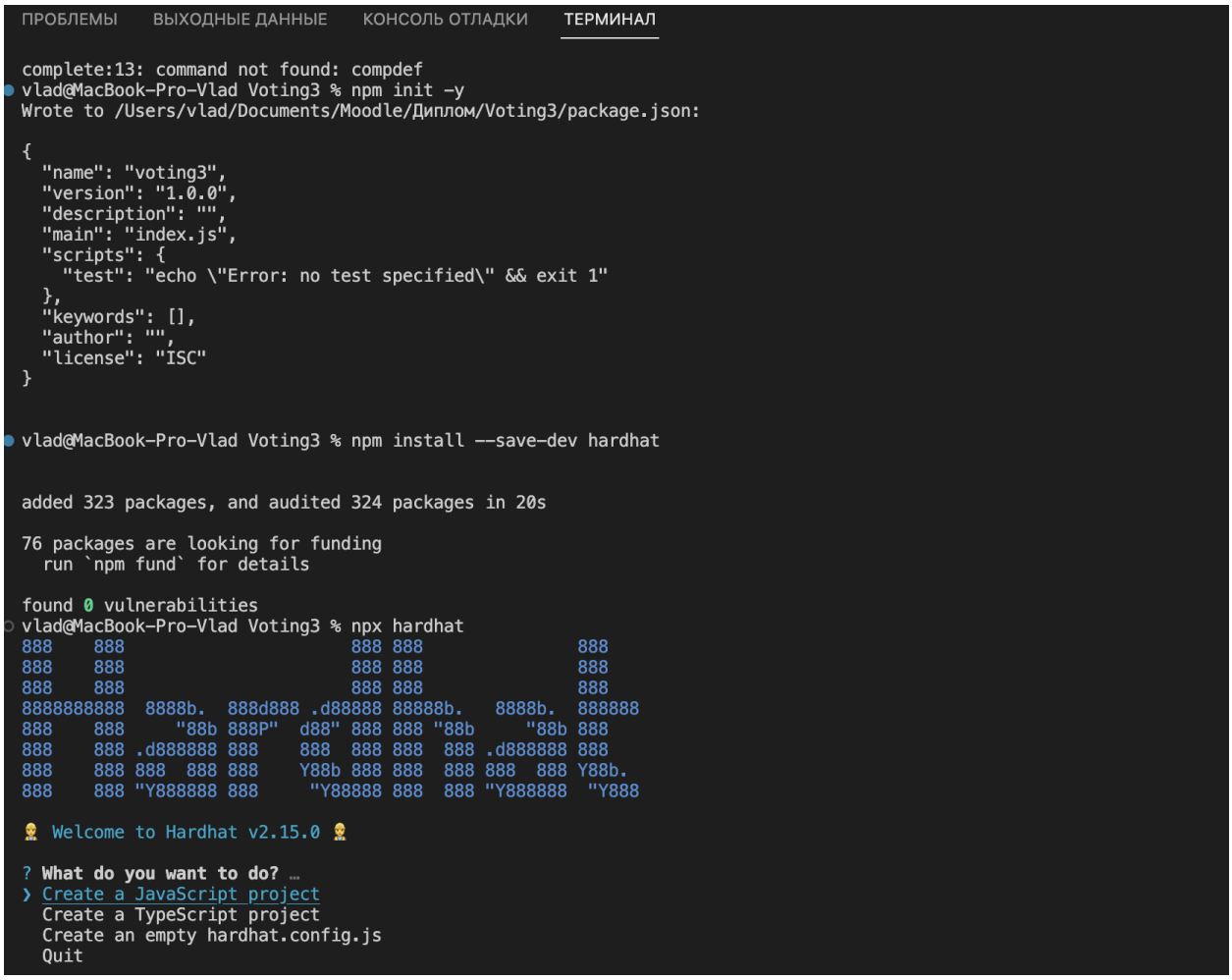

Рисунок 3.12 – Створення hardhat проєкту

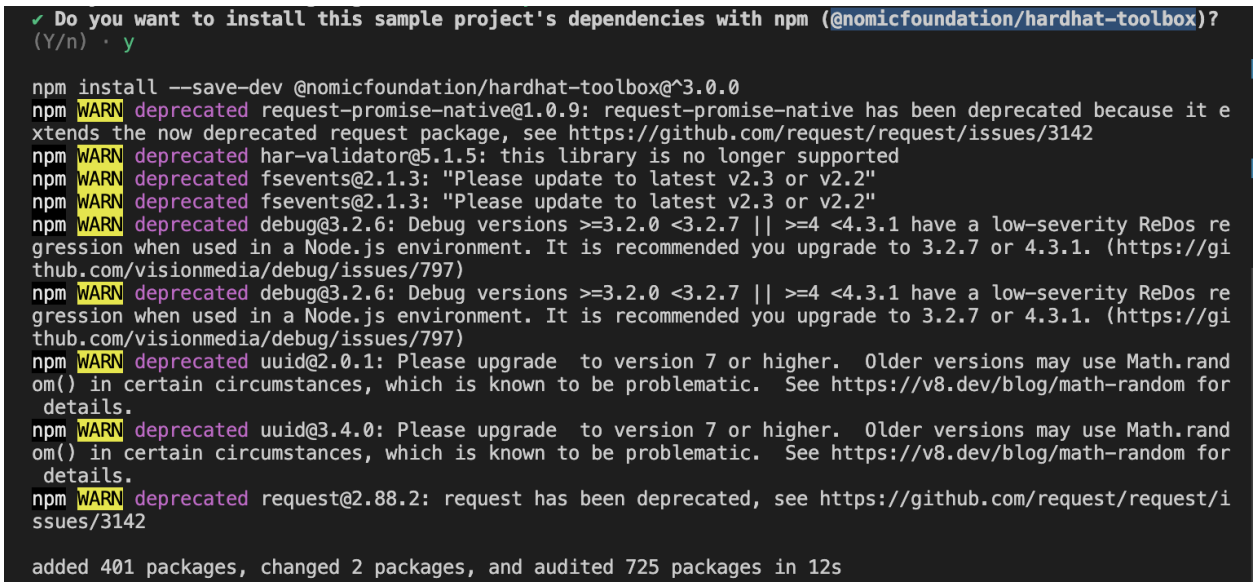

Рисунок 3.13 – Встановлення допоміжних бібліотек

3.4 Написання тестів для смарт-контракту

Використання Remix IDE було зручним на моменті написання контрактів, але коли кількість коду і функцій більшає, ми не можемо гарантувати, що не проґавимо якихось моментів. Також ми повинні враховувати різні той факт, що функції можуть діяти інакше в різних станах. Для перевірки таких станів ми і будемо використовувати тести.

Отже, ми почнемо із завантаження акаунтів із бібліотеки ethers [22] і деплою контракту в тестову мережу з наступною перевіркою цих дій (див. рис. 3.14).

```
it("Contract should be successfully deployed, account0 is owner", async function () {
      accounts = await ethers.getSigners();
      VotingContract = await ethers.getContractFactory("VotingContract");
      myVotingContract = await VotingContract.deploy(5, 5);
      await myVotingContract.deployed();
      expect(await myVotingContract.owner()).to.equal(accounts[0].address);
   });
```

Рисунок 3.14 – Тестування загрузки контракта у тестову мережу

Наступною частиною стане на перевищення можливої кількості адрес під час додавання голосування в чернетку (див. рис. 3.15).

```
it("Owner try to create voting with too many candidates", async function () {
      let candidates = new Array();
      for (i = 1; i < 19; i++) candidates.push(accounts[i].address);
      await expect(
         myVotingContract.connect(accounts[0]).addVoting(180, candidates)
      ).to.be.revertedWith("Too many candidates!");
   });
```

Рисунок 3.15 – Тест на перевищення адрес

Далі ми перевіримо можливість видаляти і додавати учасників власником контракту до того, як голосування почалося (див. рис. 3.16).

```
it("Candidate 3 deleted", async function () {
      await myVotingContract.connect(accounts[0]).deleteCandidate(0, accounts[3].address);
      const is_candidate3 = await myVotingContract.checkCandidate(0, accounts[3].address);
      expect(is_candidate3).to.equal(false);
   });
   it("Candidate 3 added again", async function () {
      await myVotingContract.connect(accounts[0]).addCandidate(0, accounts[3].address);
      const is_candidate3 = await myVotingContract.checkCandidate(0, accounts[3].address);
      expect(is_candidate3).to.equal(true);
   });
```

Рисунок 3.16 – Перевірка функцій додавання/видалення

Створимо тест для перевірки прав власника. Ніхто крім власника не повинен мати можливість видаляти учасників (див. рис. 3.17).

```
it("Candidate 3 not deleted - only owner can delete him", async function () {
      await expect( myVotingContract.connect(accounts[1]).deleteCandidate(0,
accounts[3].address)
      ).to.be.revertedWith("Error! You're not the smart contract owner!");
   });
```

Рисунок 3.17 – Перевірка модифікатора owner

До моменту старту, голосування перебуває в статусі очікування, це той самий час, коли тільки власник контракту має можливість редагувати його. Цей тест перевіряє, чи зможуть учасники проголосувати до початку голосування (див. рис. 3.18).

```
it("Nobody can't vote before start", async function () {
      const amount = new ethers.BigNumber.from(10).pow(18).mul(1);
      await expect(
         myVotingContract.connect(accounts[1]).takePartInVoting(0, accounts[3].address, {
value: amount })
      ).to.be.revertedWith("Voting not started yet");
   });
```

Рисунок 3.18 – Перевірка обмеження голосування

Створюємо перевірку на можливість почати голосування і проголосувати за учасників (див. рис. 3.19).

```
it("Voting started", async function () {
      await myVotingContract.connect(accounts[0]).startVoting(0);
      const votingInfo = await myVotingContract.getVotingInfo(0);
      //console.log(votingInfo);
      expect(votingInfo[0]).to.equal(true);
   });
it("Account 2 and 15 voted for Account 5", async function () {
      const amount = new ethers.BigNumber.from(10).pow(18).mul(1);
      await myVotingContract.connect(accounts[2]).takePartInVoting(0, accounts[5].address, {
value: amount });
      await myVotingContract.connect(accounts[15]).takePartInVoting(0, accounts[5].address,
{ value: amount });
      const votingInfo = await myVotingContract.getVotingInfo(0);
      //console.log(votingInfo);
      expect(votingInfo[5]).to.equal(accounts[5].address);
   });
```

Рисунок 3.19 – Тестування можливості голосувати

А тепер перевіримо можливість виведення призу переможцем (див. рис. 3.20).

```
it("Account 5 try to withdrow", async function () {
      await expect(
         myVotingContract.connect(accounts[5]).WithdrowMyPrize(0)
      ).to.be.revertedWith("Voting is not over yet!");
   });
```

Рисунок 3.20 – Тестування функції отримання призу та розрахунку комісії

Наразі ми написали і запустили тести і можемо приступати до деплою контракту.

3.5 Деплой смарт-контракту

Деплой смарт-контракту – це процес завантаження і запуску смарт-контракту на блокчейні. Можна написати його локально на своєму комп'ютері, але щоб він працював у мережі, потрібно завантажити його на

блокчейн. Деплой смарт-контракту – це процес, який дозволяє завантажити смарт-контракт на блокчейн і зробити його доступним для використання іншими учасниками мережі.

Деплой смарт-контракту – це важливий крок у розробці блокчейн-додатків, тому що це дає змогу створювати нові смарт-контракти та оновлювати наявні контракти, які вже були завантажені на блокчейн. Крім того, деплой смарт-контракту може використовуватися для встановлення початкових параметрів контракту.

3.5.1 Деплой у локальну тестову мережу Hardhat

Для деплою в наше локальне середовище ми повинні запустити в окремому терміналі свою ноду, створюючи тим самим дуже маленький власний блокчейн [23].

Далі ми відкриваємо ще один термінал з папки нашого проєкту, очищаємо наш проєкт від артіфактів, перекомпілюємо його і завантажуємо наш проєкт у локальне середовище за допомогою скрипта, розміщеного в папці проєкту scripts (див. рис. 3.21).

```
const { expect } = require("chai");
const hre = require("hardhat");
const ethers = hre.ethers

async function main() {
  const [signer] = await ethers.getSigners()

  const VotingContract = await ethers.getContractFactory('VotingContract', signer)
  const votingcontract = await VotingContract.deploy(10, 5)
  await votingcontract.deployed()
  console.log(transfers.address)
}
main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
```

```
  proccess.exit(1);
 })
 //npx hardhat clean
 //npx hardhat compile
 //npx hardhat run scripts/deploy.js --network localhost
```

Рисунок 3.21 – Скрипт для розгортання контракта в тестову hardhat мережу

Після цього ми перевіряємо наш блокчейн для отримання адреси контракту, пізніше ми використовуємо його для взаємодії з ним (див. рис. 3.22).

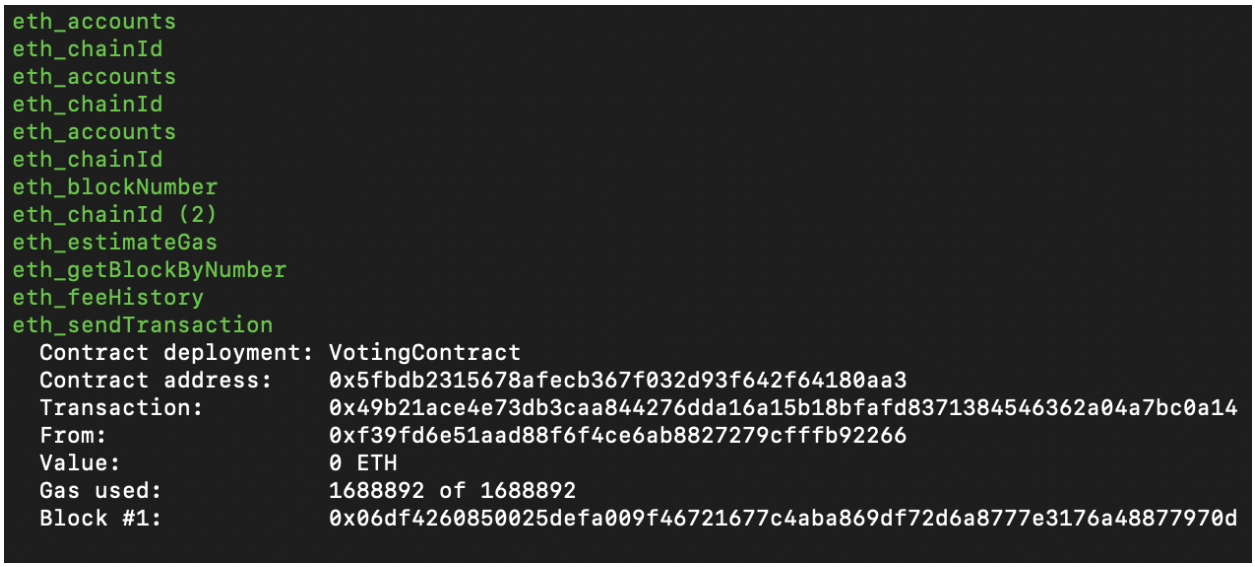

Рисунок 3.22 – Отримання інформації про розгорнутий контракт

Після чого пишемо скрипт для перевірки взаємодії з контрактом (див. рис. 3.23).

```
const hre = require("hardhat");
const ethers = hre.ethers
const VotinggContractArtifact = require('../artifacts/contracts/VotingContract.sol/VotingContract.json')
async function main() {
  const [signer] = await ethers.getSigners()
  const votingAdr = '0xe7f1725e7734ce288f8367e1bb143e90bb3f0512'
  const votingContract = new ethers.Contract(
    votingAdr,
    VotinggContractArtifact.abi,
    signer
  )
  let accounts = await ethers.getSigners();
```

```
  let candidates = new Array();
  for (i = 1; i < 3; i++) candidates.push(accounts[i].address)
  const result = await votingContract.addVoting(60,candidates)
  console.log(result)
}
main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    proccess.exit(1);
  })
//0xe7f1725e7734ce288f8367e1bb143e90bb3f0512
```

Рисунок 3.23 – Скрипт для взаємодії з нашим контрактом

Для прикладу ми одразу ж викличемо функцію додавання голосування, використовуючи першу адресу, що надається нам hardhat, адже саме ця адреса є власником і має доступ до цієї функції (див. рис. 3.24).

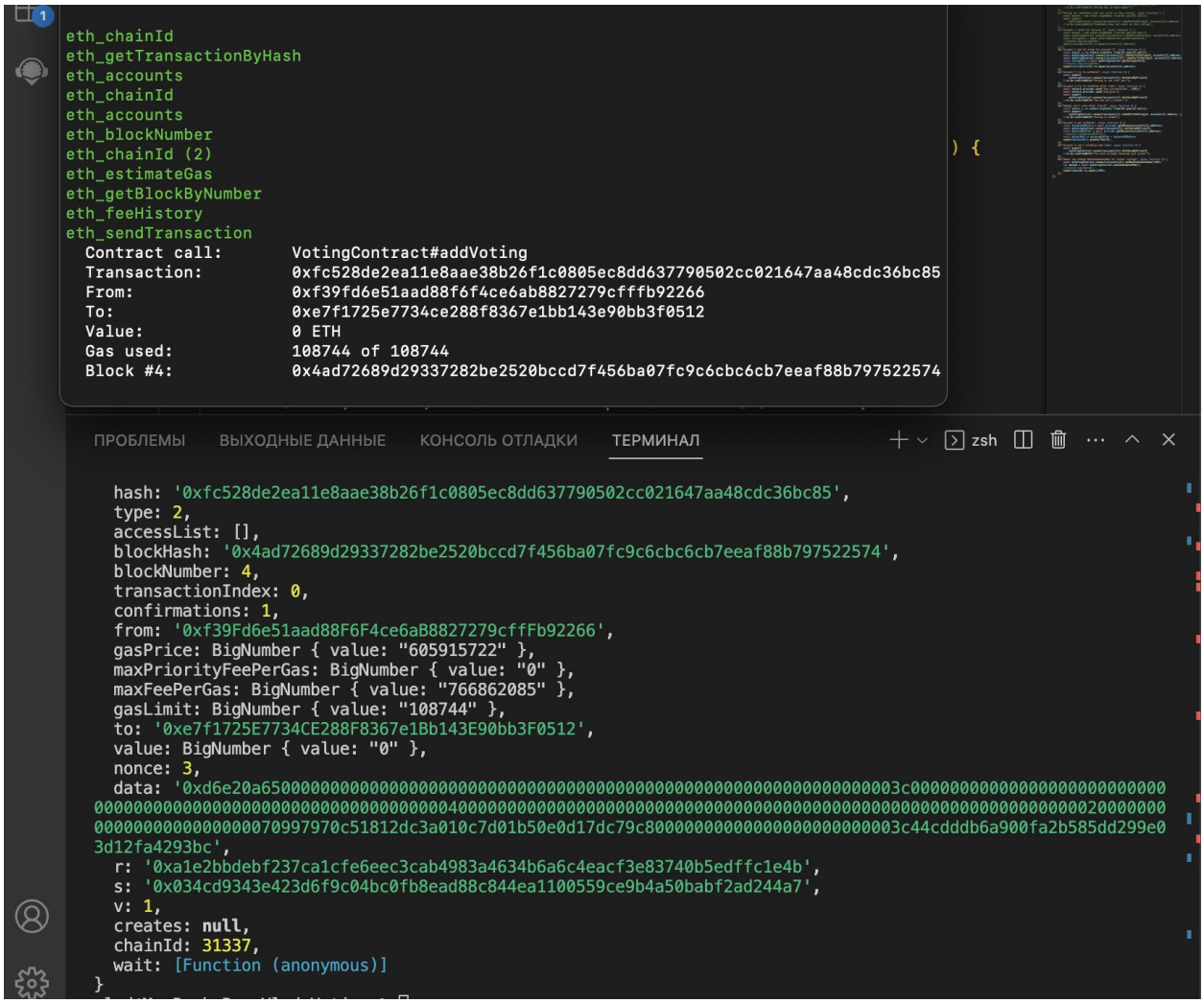

Рисунок 3.24 – Тестування взаємодії з контрактом

Як ми бачимо, помилок немає і ми навіть можемо переглянути хеш транзакції, а отже, будемо переходити до другої частини і почнемо використовувати загальнодоступну тестову мережу як середовище загальнодоступну тестову мережу Goerli.

3.5.2 Деплой у тестову мережу Goerli

Для підключення до мережі Goerli ми будемо використовувати сервіс Alchemy API [24], яку ми можемо отримати після реєстрації (див. рис. 3.25).

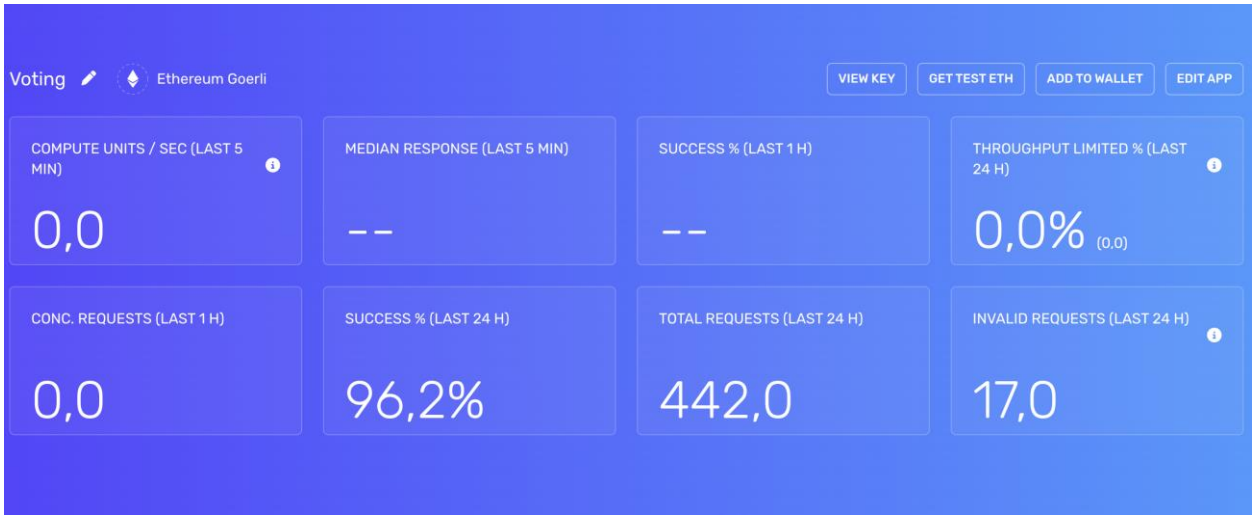

Рисунок 3.25 – Відображення взаємодії з Alchemy API

Далі ми встановлюємо гаманець Metamask, оскільки в загальній тестовій мережі нам не підійдуть адрса, що видаються hardhat.

Після встановлення ми додаємо тестову мережу Goerli і, використовуючи різноманітні крани, поповнюємо свій баланс для оплати газу і самого контракту під час розгортання.

Також ми змінюємо наш скрипт для розгортання у файлі deploy.js, signer у цьому разі змінюватиметься на нашу власну адресу, оскільки далі ми налаштуємо файл конфігурації (див. рис. 3.26).

```
require("@nomicfoundation/hardhat-toolbox");
const { expect } = require("chai");
const hre = require("hardhat");
const ethers = hre.ethers;

async function main() {
  const [signer] = await ethers.getSigners
  const VotingContract = await ethers.getContractFactory('VotingContract', signer);
  const votingcontract = await VotingContract.deploy();
  await votingcontract.deployed();
  console.log(await votingcontract.address)
}

main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error)
    process.exit(1)
  });
```

Рисунок 3.26 – Скрипт для розгортання контракта в тестову Goerli мережу

Останнє, що нам знадобиться, це змінити файл конфігурації, для цього ми створимо й під'єднаємо файл .env для прописування в ньому всієї конфіденційної інформації та додамо інформацію про мережу і наш гаманець у сам hardhat.config.js файл (див. рис. 3.27).

```
require("@nomicfoundation/hardhat-toolbox");
require("dotenv").config();
require("@nomiclabs/hardhat-etherscan");
const {ALCHEMY_KEY, PRIVATE_KEY} = process.env

const API_ALCH_KEY = ALCHEMY_KEY;
const PRV_KEY = PRIVATE_KEY;

/** @type import('hardhat/config').HardhatUserConfig */
module.exports = {
  solidity: "0.8.19",
  networks: {
    goerli: {
      url: `https://eth-goerli.g.alchemy.com/v2/0x${process.env.API_ALCH_KEY}`,
      account: [`${process.env.PRV_KEY}`],
      gas: 2100000,
```

```
    gasPrice: 8000000000,
    saveDeployments: true,
   },
   hardhat: {
    chainId: 1337
   }
  },
  gasReporter: {
   enabled: true,
   gasPrice: 2000000000
  }
};
```

Рисунок 3.27 – Налаштування файлу hardhat.config.js

Тепер ми можемо запустити скрипт і отримати адресу нашого контракту Voting (див. рис. 3.28).

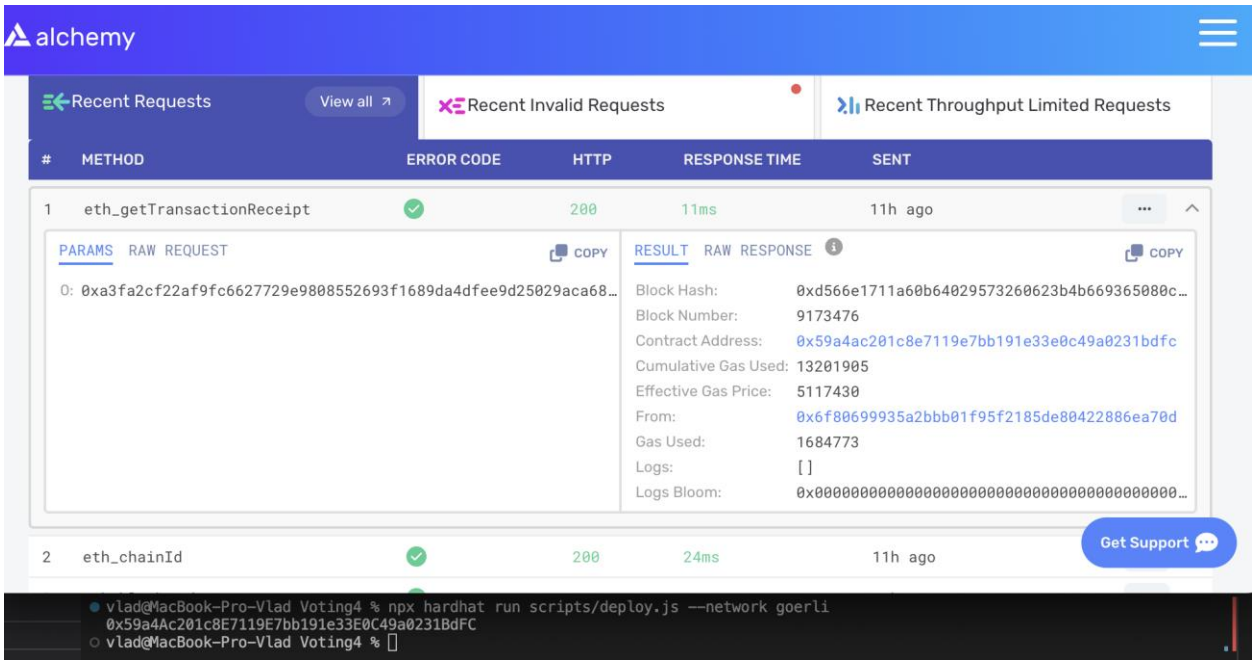

Рисунок 3.28 – Отримання інформації про наш контракт з Alchemy

Як ми бачимо, адреси збігаються, а отже, розгортання в тестову мережу Goerli пройшло успішно.

3.6 Висновки до розділу 3

У третій частині дипломної роботи було встановлено середовище розробки Remix і редактор коду Visual Studio Code. Був створений і протестований смарт-контракт, а також задеплоєний у локальну мережу hardhat і загальнодоступну мережу Goerli.

ВИСНОВКИ

У першій частині дипломної роботи було отримано знання про технологію блокчейн і способи її застосування. У роботі було висвітлено принципи взаємодії нод між собою, їхні відмінності та особливості використання. Також були отримані навички застосування інструментів розробки, перевірки та розгортання смарт-контрактів.

Під час етапу проєктування було розроблено діаграми прецендентів, функціональну діаграму та структурну схему. Було розібрано патерни проєктування Timelock, MultiSig, Commit/Reveal та Pull over Push, що позитивно вплинуло на розуміння типових алгоритмів, які використовуються в Solidity. Крім цього були розглянуті обов'язкові до вивчення типові атаки на смарт-контракти і способи їх вирішення.

У третій частині дипломної роботи з урахуванням знань, отриманих у перших двох частинах, був розроблений смарт-контракт, протестований у локальному середовищі hardhat, а також розгорнутий у загальну тестову мережу Rinkeby.

Знання, отримані під час написання моєї дипломної роботи, підвищили знання в обраній темі, а також допомогли отримати реальні навички розроблення смарт-контрактів, що допоможе мені надалі, бо я й надалі планую розвиватися в цій галузі.

ПЕРЕЛІК ПОСИЛАНЬ

1. Використання криптовалюти на ринку платежів. URL: http://visnykj.wunu.edu.ua/index.php/visnykj/article/view/706 (дата звернення: 12.03.2023).
2. Можливості використання технології blockchain у страхуванні. URL: http://visnyk-econom.uzhnu.uz.ua/archive/19_2_2018ua/24.pdf (дата звернення: 19.04.2023).
3. Застосування та аналіз технології блокчейн. URL: https://dut.edu.ua/uploads/p_421_86928920.pdf (дата звернення: 12.03.2023).
4. Теорія та практика розташування нод. URL: https://incrypted.com/ultimate-guide-po-nodam/ (дата звернення: 12.03.2023).
5. Шари блокчейна та навіщо вони потрібні. URL: https://habr.com/ru/articles/688076/ (дата звернення: 19.04.2023).
6. Опис токена ERC20. URL: https://academy.binance.com/uk/articles/an-introduction-to-erc-20-tokens (дата звернення: 12.03.2023).
7. Реалізація ERC20 стандарту. URL: https://ethereum.org/ru/developers/docs/standards/tokens/erc-20/ (дата звернення: 04.04.2023).
8. Мови програмування для створення смарт контрактів. URL: https://bytwork.com/articles/luchshie-yazyki-programmirovaniya-dlya-sozdaniya-smart-kontraktov-eth-eos-neo-i-drugie (дата звернення: 19.04.2023).
9. Документація використання середовища розробки Hardhat. URL: https://hardhat.org/hardhat-runner/docs/getting-started#overview (дата звернення: 19.04.2023).
10. Документація бібліотеки Chai. URL: https://www.chaijs.com/guide/styles/ (дата звернення: 12.03.2023).
11. Документація середовища розробки Truffle. URL:

https://trufflesuite.com/docs/ (дата звернення: 12.03.2023).

12. Smart contracts, free manual and documentation. URL: https://tinyurl.com/2v6kj9pp (дата звернення: 04.04.2023).
13. Стандарт для безпечних блокчейн-додатків OpenZeppelin. URL: https://www.openzeppelin.com/ (дата звернення: 12.03.2023).
14. Timelock Smart Contracts. URL: https://blog.chain.link/timelock-smart-contracts/ (дата звернення: 20.03.2023).
15. Build a basic multisig. URL: https://www.codementor.io/@beber89/build-a-basic-multisig-vault-in-solidity-for-ethereum-1tisbmy6ze (дата звернення: 04.04.2023).
16. Commit Reveal Scheme on Ethereum. URL: https://medium.com/gitcoin/commit-reveal-scheme-on-ethereum-25d1d1a25428 (дата звернення: 20.03.2023).
17. Pull over Push pattern. URL: https://fravoll.github.io/solidity-patterns/pull_over_push.html (дата звернення: 12.03.2023).
18. Denial of Service (DoS) Attack on Smart Contracts. URL: https://blog.finxter.com/denial-of-service-dos-attack-on-smart-contracts/ (дата звернення: 12.03.2023).
19. Що таке Re-entrancy attack? URL: https://habr.com/ru/articles/655639/ (дата звернення: 20.03.2023).
20. Документація Remix. URL: https://remix.run/docs/en/main (дата звернення: 12.03.2023).
21. Економія газу в Ethereum. URL: https://tinyurl.com/ykm7284z (дата звернення: 04.04.2023).
22. Ethers documentation. URL: https://docs.ethers.org/v5/ (дата звернення: 12.03.2023).
23. Документація по розгортанню у мережу Hardhat. URL: https://hardhat.org/hardhat-runner/docs/guides/deploying (дата звернення: 20.03.2023).
24. Alchemy deploy guide. URL: https://docs.alchemy.com/docs/hello-world-smart-contract (дата звернення: 04.04.2023).

ДОДАТОК А

Смарт-контракт системи голосування

```
/ SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

contract VotingContract {
    address public owner;
    uint256 public counter;
    uint256 public maxCandidatesNum;

    struct Candidate {
        uint256 balance;
        bool isExistOnThisVoting; //перевірка на наявність у списку голосуванні
    }

    struct Voting {
        bool started;
        address Winner;
        uint256 StartDate;
        uint256 WinnerBalance;
        uint256 Bank;
        uint256 Period;
        mapping(address => Candidate) Candidates;
    }

    mapping(uint256 => Voting) private Votings; // смешанный, не может быть public

    uint8 public immutable Comission;

    constructor(/*uint256 _maxCandidatesNum, uint8 _comission*/) {
        owner = msg.sender;
        Comission = 5;//_comission;
        maxCandidatesNum = 5;//_maxCandidatesNum;
    }

    function takePartInVoting(uint8 _votingID, address _candidate)
        public
        payable
    {
        require(Votings[_votingID].started, "Voting not started yet");
        require(
            Votings[_votingID].StartDate + Votings[_votingID].Period >
                block.timestamp,
            "Voting is ended"
        );
        require(
            checkCandidate(_votingID, _candidate),
            "Candidate does not exist on this voting"
```

```
    );
    Votings[_votingID].Candidates[_candidate].balance += msg.value;
    Votings[_votingID].Bank += msg.value;
    //перевіряємо чи не змінився лідер після кожного нового голосу
    if (
        Votings[_votingID].Candidates[_candidate].balance >
        Votings[_votingID].WinnerBalance
    ) {
        Votings[_votingID].WinnerBalance = Votings[_votingID].Candidates[_candidate].balance;
        Votings[_votingID].Winner = _candidate;
    }
}

function WithdrowMyPrize(uint256 _votingID) public {
    require(Votings[_votingID].started, "Voting not started yet");
    require(
        Votings[_votingID].StartDate + Votings[_votingID].Period <
            block.timestamp,
        "Voting is not over yet!"
    );
    require(
        msg.sender == Votings[_votingID].Winner,
        "You are not a winner!"
    );
    require(
        Votings[_votingID].Bank > 0,
        "You have already received your prize!"
    );
    uint256 amount = Votings[_votingID].Bank;
    uint256 ownersComission = (Comission * amount) / 100; //вишукуємо нашу комісію
    uint256 clearAmount = amount - ownersComission;
    Votings[_votingID].Bank = 0; //захист від reentrancy attack
    payable(owner).transfer(ownersComission);
    payable(msg.sender).transfer(clearAmount);
}

function getVotingInfo(uint256 _votingID)
    public
    view
    returns (
        bool,
        uint256,
        uint256,
        uint256,
        uint256,
        address
    )
{
    return (
        Votings[_votingID].started,
        Votings[_votingID].StartDate,
        Votings[_votingID].Period,
        Votings[_votingID].WinnerBalance,
```

```
        Votings[_votingID].Bank,
        Votings[_votingID].Winner
    );
}

function checkCandidate(uint256 _votingID, address _candidate)
    public
    view
    returns (bool)
{
    return (Votings[_votingID].Candidates[_candidate].isExistOnThisVoting);
}

function addVoting(uint256 _period, address[] calldata _candidates) // созд. голосование с периодом
    public
    onlyOwner
{
    require(_candidates.length < maxCandidatesNum, "Too many candidates!");
    Votings[counter].Period = _period;
    for (uint256 i = 0; i < _candidates.length; i++) {
        addCandidate(counter, _candidates[i]);
    }
    emit votingDraftCreated(counter);
    counter++;
}

function startVoting(uint256 _votingID) public onlyOwner {
    Votings[_votingID].started = true;
    Votings[_votingID].StartDate = block.timestamp;
    emit votingStarted(_votingID, block.timestamp);
}

function editVotingPeriod(uint256 _votingID, uint256 _newPeriod) //смена периода
    public
    onlyOwner
{
    require(
        Votings[_votingID].started == false,
        "Voting has already begun!"
    );
    Votings[_votingID].Period = _newPeriod;
}

function addCandidate(uint256 _votingID, address _candidate)
    public
    onlyOwner
{
    require(
        Votings[_votingID].started == false,
        "Voting has already begun!"
    );
    Votings[_votingID].Candidates[_candidate].isExistOnThisVoting = true;
    emit candidateInfo(_votingID, _candidate, true);
```

```
    }

    function deleteCandidate(uint256 _votingID, address _candidate)
        public
        onlyOwner
    {
        require(
            Votings[_votingID].started == false,
            "Voting has already begun!"
        );
        Votings[_votingID].Candidates[_candidate].isExistOnThisVoting = false;
        emit candidateInfo(_votingID, _candidate, false);
    }

    function setMaxCandidatesNum(uint256 _maxCandidatesNum) public onlyOwner {
        maxCandidatesNum = _maxCandidatesNum;
    }

    modifier onlyOwner() {
        require(
            msg.sender == owner,
            "Error! You're not the smart contract owner!"
        );
        _;
    }

    event candidateInfo(
        uint256 indexed votingID,
        address indexed candidate,
        bool existOnThisVoting
    );
    event votingDraftCreated(uint256 indexed votingID);
    event votingStarted(uint256 indexed votingID, uint256 startDate);
}
```

ДОДАТОК Б

Тести для перевірки роботи контракту

```
const { expect } = require("chai");
const { ethers } = require("hardhat");
let accounts;
let myVotingContract;
const provider = ethers.provider;

describe("VotingContract", function () {
  it("Contract should be successfully deployed, account 0 is owner", async function () {
    accounts = await ethers.getSigners();
    VotingContract = await ethers.getContractFactory("VotingContract");
    myVotingContract = await VotingContract.deploy(10, 5);
    await myVotingContract.deployed();
    expect(await myVotingContract.owner()).to.equal(accounts[0].address);
  });
  it("Owner try to create voting with too many candidates", async function () {
    let candidates = new Array();
    for (i = 1; i < 19; i++) candidates.push(accounts[i].address);
    await expect(
      myVotingContract.connect(accounts[0]).addVoting(180, candidates)
    ).to.be.revertedWith("Too many candidates!");
  });
  it("Owner created a vote, the counter is increased", async function(){
    const counter_before = await myVotingContract.counter();
    let candidates= new Array();
    for (i = 1; i < 10; i++) candidates.push(accounts[i].address);
    await myVotingContract.connect(accounts[0]).addVoting(180, candidates);
    const counter_after = await myVotingContract.counter();
    expect(counter_after-counter_before).to.equal(1);
    const is_candidate5 = await myVotingContract.checkCandidate(counter_before, accounts[5].address);
    expect(is_candidate5).to.equal(true);
  });
  it("Owner created another voting", async function () {
    const counter_before = await myVotingContract.counter();
    let candidates = new Array();
    for (i = 1; i < 4; i++) candidates.push(accounts[i].address);
    await myVotingContract.connect(accounts[0]).addVoting(180, candidates);
    const is_candidate5 = await myVotingContract.checkCandidate(counter_before, accounts[5].address);
    expect(is_candidate5).to.equal(false);
  });
  it("Candidate 3 deleted", async function () {
    await myVotingContract.connect(accounts[0]).deleteCandidate(0, accounts[3].address);
    const is_candidate3 = await myVotingContract.checkCandidate(0, accounts[3].address);
    expect(is_candidate3).to.equal(false);
  });
  it("Candidate3 added again", async function () {
    await myVotingContract.connect(accounts[0]).addCandidate(0, accounts[3].address);
```

```
    const is_candidate3 = await myVotingContract.checkCandidate(0, accounts[3].address);
    expect(is_candidate3).to.equal(true);
});
it("Candidate 3 not deleted - only owner can delete him", async function () {
    await expect(
        myVotingContract.connect(accounts[1]).deleteCandidate(0, accounts[3].address)
    ).to.be.revertedWith("Error! You're not the smart contract owner!");
});
it("Owner changed voting period", async function () {
    await myVotingContract.connect(accounts[0]).editVotingPeriod(0,190);
    const votingInfo = await myVotingContract.getVotingInfo(0);
    //console.log(votingInfo);
    expect(votingInfo[2]).to.equal(190);
});
it("Nobody can't vote before start", async function () {
    const amount = new ethers.BigNumber.from(10).pow(18).mul(1);
    await expect(
        myVotingContract.connect(accounts[1]).takePartInVoting(0, accounts[3].address, { value: amount })
    ).to.be.revertedWith("Voting not started yet");
});
it("Voting started", async function () {
    await myVotingContract.connect(accounts[0]).startVoting(0);
    const votingInfo = await myVotingContract.getVotingInfo(0);
    //console.log(votingInfo);
    expect(votingInfo[0]).to.equal(true);
});
it("Account 2 and 15 voted for Account 5", async function () {
    const amount = new ethers.BigNumber.from(10).pow(18).mul(1);
    await myVotingContract.connect(accounts[2]).takePartInVoting(0, accounts[5].address, { value: amount });
    await myVotingContract.connect(accounts[15]).takePartInVoting(0, accounts[5].address, { value: amount });
    const votingInfo = await myVotingContract.getVotingInfo(0);
    //console.log(votingInfo);
    expect(votingInfo[5]).to.equal(accounts[5].address);
});
it("Account 5 try to withdrow", async function () {
    await expect(
        myVotingContract.connect(accounts[5]).WithdrowMyPrize(0)
    ).to.be.revertedWith("Voting is not over yet!");
});
it("Account 4 try to withdrow after time", async function () {
    await network.provider.send("evm_increaseTime", [200]);
    await network.provider.send("evm_mine");
    await expect(
        myVotingContract.connect(accounts[4]).WithdrowMyPrize(0)
    ).to.be.revertedWith("You are not a winner!");
});
it("Nobody can't vote after finish", async function () {
    const amount = new ethers.BigNumber.from(10).pow(18).mul(1);
    await expect(
        myVotingContract.connect(accounts[1]).takePartInVoting(0, accounts[3].address, { value: amount })
    ).to.be.revertedWith("Voting is ended");
});
it("Account 5 got withdrow", async function () {
```

```
    const balanceH2Before = await provider.getBalance(accounts[5].address);
    await myVotingContract.connect(accounts[5]).WithdrowMyPrize(0);
    const balanceH2After = await provider.getBalance(accounts[5].address);
    //console.log(balanceH2After);
    const balanceDif = balanceH2After - balanceH2Before;
    expect(balanceDif).greaterThan(0);
  });
  it("Account 5 can't withdrow 2nd time", async function () {
    await expect(
      myVotingContract.connect(accounts[5]).WithdrowMyPrize(0)
    ).to.be.revertedWith("You have already received your prize!");
  });
  it("Owner can change MaxCandidatesNum for futher votings", async function () {
    await myVotingContract.connect(accounts[0]).setMaxCandidatesNum(1500);
    let maxnum = await myVotingContract.maxCandidatesNum();
    //console.log(maxnum);
    expect(maxnum).to.equal(1500);
  });
});
```

ДОДАТОК В

Скрипт розгортання контракту

```
require("@nomicfoundation/hardhat-toolbox");
const { expect } = require("chai");
const hre = require("hardhat");
const ethers = hre.ethers;

async function main() {
  const [signer] = await ethers.getSigners()//process.env.PRIVATE_KEY//[process.env.PRIVATE_KEY]
  const VotingContract = await ethers.getContractFactory('VotingContract', signer);
  const votingcontract = await VotingContract.deploy();
  await votingcontract.deployed();
  console.log(await votingcontract.address)
}

main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error)
    process.exit(1)
  });

  //npx hardhat clean
  //npx hardhat compile
  //npx hardhat run scripts/deploy.js --network localhost
```