НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

УДК

«ПОГОДЖЕНО»	**«ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ»**
Декан факультету інформаційних технологій	Завідувач кафедри комп'ютерних наук
Болбот І.М., д.т.н., професор	Голуб Б.Л., к.т.н., доцент
_____	_____
____ _____ 2024 р.	**____ _____ 2024 р.**

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему __ Система управління смарт контрактами з використанням технологій блокчейн 004.77:004.9:336_

Спеціальність ___122 - Комп'ютерні науки_____
(код і назва)

Освітня програма ____Інформаційно управляючі системи та технології_
(назва)

Орієнтація освітньої програми ____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми

_____ _____ ___Голуб Б.Л.__
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник магістерської кваліфікаційної роботи

_____д.т.н., професор_____ _____ _____Хиленко В.В._____
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ _____Ващук Н.В._____
(підпис) (ПІБ студента)

КИЇВ - 2024

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) Інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук

к.т.н., доцент _____ Голуб Б.Л.
(науковий ступінь, вчене звання) (підпис) (ПІБ)
"____" _____ 2024 року

ЗАВДАННЯ

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Ващук Назар Віталійович
(прізвище, ім'я, по батькові)

Спеціальність 122 - Комп'ютерні науки
(код і назва)

Освітня програма Інформаційно управляючі системи та технології
(назва)

Орієнтація освітньої програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи Система управління смарт контрактами з використанням технологій блокчейн
затверджена наказом ректора НУБіП України від "____"_____20___р. №_____

Термін подання завершеної роботи на кафедру _____
(рік, місяць, число)

Вихідні дані до магістерської кваліфікаційної роботи включають аналіз технологій блокчейн, розробку та впровадження смарт-контрактів для автоматизації фінансових операцій, інтеграцію з біржовими джерелами даних та створення механізмів прогнозування змін цін.

Перелік питань, що підлягають дослідженню:

1. Системний аналіз предметної області
2. Моделювання системи
3. Розробка системи
4. Результати дослідження

Перелік графічного матеріалу (за потреби) _____

Дата видачі завдання "_____"_____20___ р.

Керівник магістерської кваліфікаційної роботи _____ Хиленко В.В.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання _____ Ващук Н.В.
(підпис) (прізвище та ініціали студент

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

1. PoC (Proof of Capacity) — алгоритм консенсусу, що використовує доступний обсяг пам’яті, а не обчислювальну потужність, для верифікації транзакцій і майнінгу блоків.
2. PoW (Proof of Work) — алгоритм консенсусу, що базується на розв’язанні складних обчислювальних задач, потребує значних ресурсів і витрат електроенергії.
3. NFT (Non-Fungible Token) — унікальний токен, що представляє право власності на конкретний цифровий або фізичний актив і не підлягає взаємозаміні.
4. API (Application Programming Interface) — інтерфейс прикладного програмування, що дозволяє програмам взаємодіяти між собою.
5. GUI (Graphical User Interface) — графічний інтерфейс користувача, який забезпечує взаємодію користувача із системою через візуальні елементи.
6. SDK (Software Development Kit) — набір інструментів для розробки програмного забезпечення, що включає бібліотеки, документацію та зразки коду.
7. IDE (Integrated Development Environment) — інтегроване середовище розробки, що об’єднує інструменти для написання, компіляції та тестування програмного коду.
8. JDK (Java Development Kit) — набір інструментів для розробки програм на мові Java, що включає компілятор, бібліотеки, інструменти для налагодження та інше.
9. BouncyCastle — криптографічна бібліотека для Java, яка забезпечує реалізацію криптографічних алгоритмів і використовується для захисту даних.
10. Gradle — інструмент для автоматизації збирання проєктів, що полегшує управління залежностями, компіляцію та тестування.
11. Smart Contract — смарт-контракт, програма, яка автоматично виконує умови договору в блокчейні, забезпечуючи прозорість та незмінність.

12. Bytecode — байт-код, проміжний код, що виконується віртуальною машиною, наприклад, Java Virtual Machine (JVM).

13. Hash — хеш-функція, криптографічна функція, що генерує фіксований розмір вихідного рядка (хеш) з вхідних даних, використовується для захисту та перевірки цілісності даних.

14. Signum — блокчейн-платформа, яка використовує PoC для верифікації транзакцій і забезпечення безпеки, є основою для побудови системи у дипломному проєкті.

15. Emulator — емулятор, програмний інструмент для імітації роботи системи або її окремих компонентів у тестовому середовищі.

16. Rinkeby/Ropsten — тестові мережі блокчейну Ethereum, що використовуються для перевірки та тестування смарт-контрактів перед їх розгортанням у основній мережі.

17. Tx (Transaction) — транзакція, передача даних або цінностей між учасниками блокчейн-мережі, записується в блокчейн як запис, що підтверджує обмін.

18. SHA (Secure Hash Algorithm) — безпечний хеш-алгоритм, що використовується для генерації унікальних хешів даних, які забезпечують цілісність інформації.

ВСТУП

Актуальність теми дослідження. Сучасний розвиток інформаційних технологій, особливо в галузі децентралізованих систем, сприяє стрімкому зростанню популярності блокчейн-технологій, які знаходять широке застосування у різних галузях – від фінансових операцій до автоматизації управління інтелектуальними контрактами. Проте існуючі рішення часто мають обмеження, пов’язані з безпекою, масштабованістю та ефективністю. Зокрема, актуальним є питання ефективного управління смарт-контрактами з урахуванням конфіденційності даних, прозорості та децентралізації. Впровадження інноваційних підходів на базі блокчейн дозволяє вирішувати ці проблеми та покращувати загальну ефективність систем управління смарт-контрактами. Тому дослідження системи управління смарт-контрактами з використанням блокчейн-технологій є надзвичайно актуальним для сучасної інформаційної інфраструктури.

Об’єктом дослідження є процес управління смарт-контрактами, які використовуються для автоматизації взаємодій між учасниками децентралізованих систем.

Предметом дослідження є система управління смарт-контрактами, що реалізована на базі технологій блокчейн, з використанням алгоритмів для підвищення ефективності управління процесами у децентралізованих системах.

Мета дослідження полягає у розробці системи управління смарт-контрактами з використанням технологій блокчейн, що дозволить підвищити рівень безпеки, прозорості та ефективності обміну даними між учасниками системи.

Для досягнення цієї мети в дослідженні поставлено такі завдання:

- провести аналіз існуючих систем управління смарт-контрактами та виявити їхні недоліки;
- розробити вимоги до нової системи управління смарт-контрактами з використанням технологій блокчейн;

- створити архітектуру системи, яка враховує специфіку децентралізованого середовища та високі вимоги до безпеки;
- реалізувати прототип системи та протестувати його на відповідність заданим вимогам;
- сформулювати рекомендації щодо подальшого впровадження та масштабування системи.

Методи дослідження включають аналіз літератури та існуючих рішень у галузі блокчейн та смарт-контрактів, розробку системної архітектури на основі принципів децентралізованих систем, програмну реалізацію за допомогою мов програмування та фреймворків для блокчейн-платформ, а також тестування та верифікацію результатів у тестовому середовищі.

Наукова новизна дослідження полягає в тому, що вперше розроблено систему управління смарт-контактами з використанням інноваційних підходів до безпеки та ефективності, запропоновано удосконалення архітектури управління децентралізованими контрактами та алгоритмів перевірки даних в умовах блокчейн середовища.

Апробація результатів дослідження була проведена в рамках науково-технічної конференції «ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ: ЕКОНОМІКА, ТЕХНІКА, ОСВІТА», що сприяло отриманню фахових відгуків і пропозицій для подальшого вдосконалення системи.

Структура магістерської роботи складається з вступу, чотирьох розділів, висновків, списку використаних джерел та додатків.

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис смарт-контрактів і блокчейн-технологій.

Смарт-контракти є одним із ключових елементів сучасних децентралізованих систем. Можна сказати, що це самовиконувані програми, які автоматично виконують певні умови угоди, закодовані у програмний код [1, с. 34]. Смарт-контракт, за своєю суттю, є умовною програмою, яка виконує транзакції між сторонами, якщо виконано певні попередньо визначені умови. Вперше термін "смарт-контракт" був введений криптографом Ніком Сабо у 1994 році [1, с. 67]. Однак значну популярність ця технологія набула з розвитком блокчейн-технологій, зокрема завдяки платформі Ethereum, яка дозволила створювати та виконувати смарт-контракти на основі децентралізованих мереж [1, с. 102].

Основною перевагою смарт-контрактів є їхня автономність і прозорість [1, с. 28]. Смарт-контракти діють без участі посередників або контрольних органів, що робить процес виконання угод більш надійним і швидким. Наприклад, як тільки визначені умови контракту виконуються, смарт-контракт автоматично запускає передбачені дії. Це знижує ризик шахрайства та маніпуляцій з боку третіх осіб [5, с. 39], а також зменшує витрати на транзакції, оскільки відсутність посередників суттєво скорочує витрати на управління угодами [2, с. 45].

Смарт-контракти можна використовувати у багатьох галузях, наприклад можна взяти фінанси, де автоматизація кредитних угод допомагає прискорити процес взаємодії між позичальниками і кредиторами [2, с. 63]; також чудовим прикладом є право, де смарт-контракти можуть використовуватись для реєстрації прав власності та забезпечення виконання юридичних угод [2, с. 92]; ще чудовим прикладом є логістика, де за допомогою смарт-контрактів можна відстежувати рух товарів і оптимізувати ланцюжки поставок [2, с. 77]; а також страхування, де автоматизовані виплати за страховими полісами допомагають

підвищити ефективність обробки запитів та виключити людський фактор [2, с. 33].

Блокчейн є основною технологією, яка забезпечує функціонування смарт-контрактів. Блокчейн представляє собою децентралізовану розподілену базу даних, яка зберігає всі транзакції у вигляді ланцюжка блоків [2, с. 55]. Кожен блок містить інформацію про попередні транзакції, що гарантує цілісність і незмінність даних. Це досягається за допомогою криптографічних методів шифрування та хешування, що робить блокчейн стійким до змін і атак [2, с. 88].

Однією з головних характеристик блокчейну є його децентралізація. Учасники мережі мають рівний доступ до загальної бази даних, а всі рішення щодо змін у мережі приймаються консенсусом [2, с. 103]. Це усуває потребу в центральному керуючому органі та значно підвищує безпеку даних. Іншою важливою особливістю є прозорість, адже кожен учасник має змогу перевірити всі транзакції, збережені в блоках [2, с. 132]. Незмінність даних у блокчейні гарантує, що інформація, додана в блок, не може бути змінена або видалена без погодження всієї мережі [2, с. 57]. Це робить блокчейн ідеальним інструментом для збереження критично важливих даних, таких як фінансові операції, юридичні угоди або медичні записи.

Завдяки поєднанню блокчейн-технологій і смарт-контрактів створюється новий рівень довіри між учасниками процесу, який не залежить від центральних посередників або регуляторів [2, с. 69]. Це робить ці технології незамінними у сучасній цифровій економіці.

Публічні блокчейни, такі як Bitcoin та Ethereum, працюють у відкритій мережі, де будь-хто може брати участь, підтверджувати транзакції та переглядати дані. Це забезпечує високу прозорість, але також потребує потужних механізмів захисту від зловмисників. Приватні блокчейни, навпаки, створюються для внутрішнього використання організаціями, де доступ до мережі контролюється та обмежується лише певними учасниками. Такі блокчейни надають більший контроль над даними і транзакціями, але при цьому жертвують частиною децентралізації та прозорості. Приватні блокчейни

надають компаніям або організаціям можливість контролювати, хто має право доступу до даних і виконання транзакцій, що підвищує конфіденційність та швидкість обробки. Характеристику блокчейн-технологій розглянемо детальніше у таблиці 1.1.

Таблиця 1.1

Характеристика блокчейн-технологій

Характеристика	Опис
Децентралізація	Блокчейн не має централізованого органу або сервера. Всі учасники мережі мають рівний доступ до інформації, що забезпечує стійкість до атак та втручань третіх осіб.
Прозорість і незмінність	Всі транзакції є публічними, і будь-хто може перевірити їхню достовірність. Після додавання даних до блокчейну їх неможливо змінити або видалити, що захищає від маніпуляцій.
Безпека	Використання криптографії забезпечує захист даних від несанкціонованого доступу. Кожен блок має криптографічний підпис, що зв'язує його з попереднім блоком, створюючи безпечний ланцюг.
Смарт-контракти	Смарт-контракти автоматизують виконання угод між сторонами без участі посередників. Контракти виконуються автоматично при досягненні визначених умов.
Незмінність (Immutability)	Після того, як дані додані до блокчейну, вони не можуть бути змінені або видалені. Це гарантує постійну історію всіх транзакцій та високу надійність даних.
Анонімність	Учасники можуть взаємодіяти з блокчейном без необхідності розкривати свою особистість. Це забезпечує конфіденційність і захист персональних даних.
Консенсусні алгоритми	Для підтвердження транзакцій блокчейн використовує різні механізми консенсусу (Proof of Work, Proof of Stake), які забезпечують єдину версію даних у мережі.
Транспарентність	Всі учасники можуть переглядати транзакції в режимі реального часу, що забезпечує високий рівень прозорості та запобігає шахрайству.

Узагальнюючи, взаємодія смарт-контрактів та блокчейн-технологій забезпечує високий рівень надійності та безпеки під час виконання угод, що

робить ці технології важливими інструментами у багатьох сферах сучасної економіки та управління.

1.2 Аналіз існуючих рішень щодо управління смарт-контрактами.

Сучасні рішення для управління смарт-контрактами включають такі платформи, як Ethereum, Hyperledger Fabric, EOS і Tezos. Кожна з них пропонує унікальні підходи до виконання смарт-контрактів, забезпечуючи різні рівні децентралізації, масштабованості та безпеки.

Ethereum є найбільш відомою та популярною платформою для управління смарт-контрактами. Її децентралізована природа дозволяє розробникам створювати смарт-контракти на основі блокчейн і запускати децентралізовані додатки (dApps). Основною перевагою Ethereum є його потужна екосистема і велика кількість розробників, які підтримують та розвивають платформу. Недоліки включають високу вартість транзакцій (Gas fees) та проблеми з масштабованістю через обмежену пропускну здатність мережі. На рис. 1.1. можемо побачити детальніше процес виконання смарт-контрактів

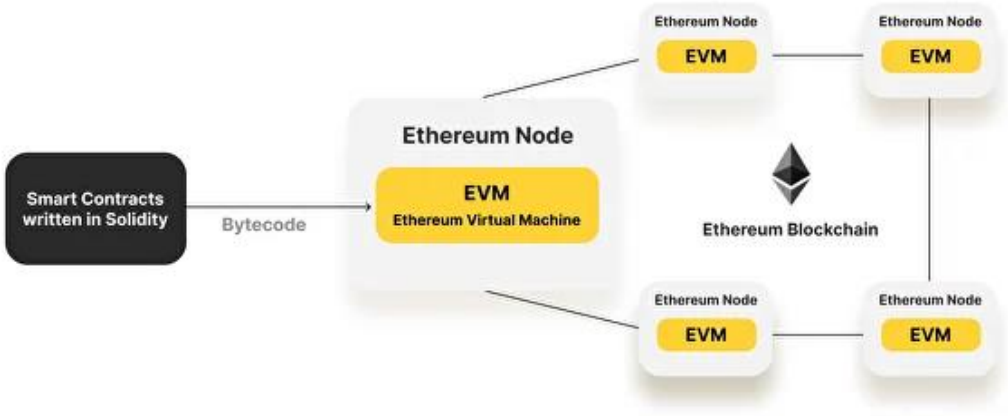

Рис.1.1 Процес виконання смарт-контрактів

Смарт-контракти пишуться мовою програмування Solidity і компілюються в байт-код. Цей байт-код передається на вузли Ethereum (Ethereum Nodes), які мають вбудовану Ethereum Virtual Machine (EVM). EVM інтерпретує та виконує

байт-код смарт-контракту. Кожен вузол у мережі Ethereum має копію блокчейну, і EVM на кожному вузлі виконує одну і ту саму операцію, що забезпечує узгодженість виконання контрактів по всій децентралізованій мережі.

Далі розглянемо Hyperledger Fabric яка є приватною блокчейн-платформою, розроблена для корпоративного використання під егідою Linux Foundation у рамках проєкту Hyperledger. Відмінною рисою Fabric є її модульна архітектура, яка дозволяє налаштовувати мережу відповідно до специфічних вимог бізнесу. Детальніше розглянемо архітектуру на рис. 1.2.

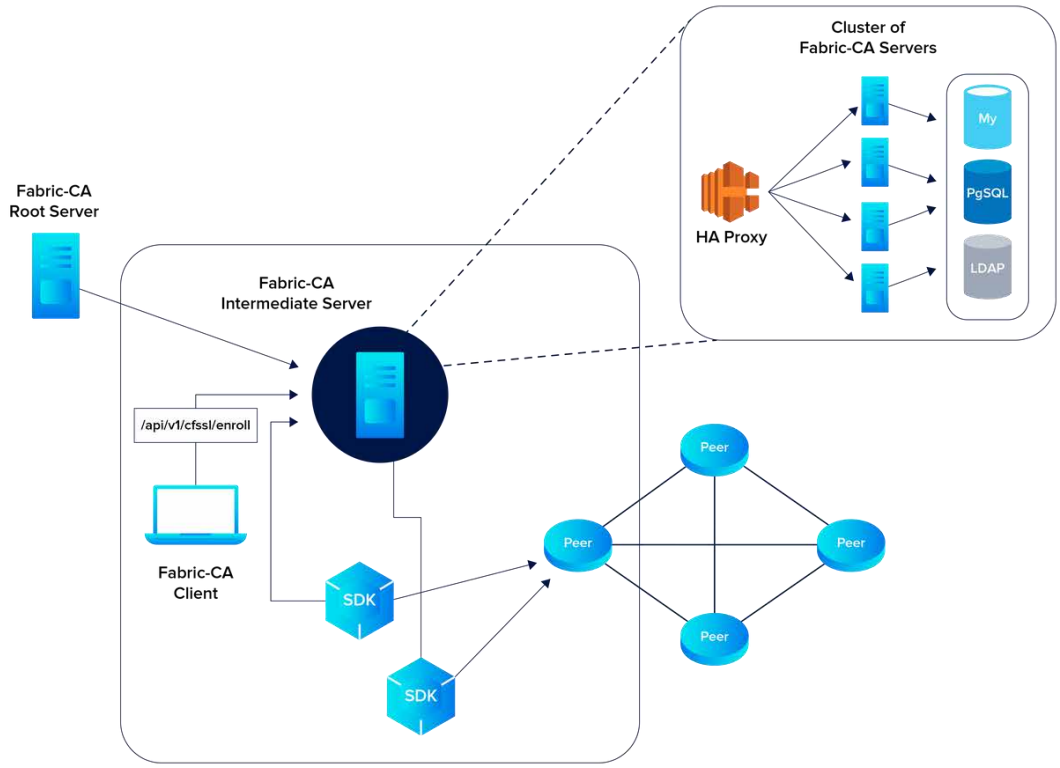

Рис.1.2 Архітектура Hyperledger Fabric

Архітектура включає кореневий сервер сертифікації (Fabric-CA Root Server), який є центральним елементом для видачі та управління сертифікатами і ключами учасників мережі. Проміжний сервер сертифікації (Fabric-CA Intermediate Server) отримує сертифікати від кореневого сервера і передає їх користувачам або іншим вузлам у мережі. Клієнти, які взаємодіють із сертифікаційним центром через Fabric-CA Client, можуть отримувати необхідні сертифікати для участі у мережі.

Ключову роль у мережі грають вузли-учасники (Peer Nodes), які зберігають копії блокчейну і виконують смарт-контракти (chaincode). Компонент Orderer

відповідає за впорядкування транзакцій перед їх додаванням до блокчейну, забезпечуючи консенсус у мережі. Додатково на зображенні представлено кластер серверів сертифікаційного центру, який забезпечує надійність та балансування навантаження, використовуючи HA Proxy.

Однією з ключових особливостей є підтримка смарт-контрактів, які тут називаються "Chaincode". Chaincode може бути написаний різними мовами програмування, такими як Go або Java, що робить цю платформу гнучкішою у порівнянні з іншими рішеннями, де підтримуються лише певні мови.

Основною перевагою Hyperledger Fabric є гнучкість та конфіденційність, що дозволяє налаштовувати різні рівні доступу для учасників мережі. Це досягається завдяки механізму приватних каналів, які дозволяють групам учасників взаємодіяти та обмінюватися конфіденційними даними без доступу сторонніх осіб. Таким чином, різні учасники можуть мати різний рівень доступу до даних і функцій мережі, що є критично важливим для корпоративних систем, де конфіденційність та контроль мають велике значення.

Fabric також підтримує плюралістичну модель консенсусу, що дозволяє використовувати різні механізми підтвердження транзакцій залежно від потреб компанії, наприклад, такі як Practical Byzantine Fault Tolerance (PBFT) або Raft. Це дозволяє досягти високої продуктивності та масштабованості системи, що особливо важливо для корпоративних застосувань з великими обсягами транзакцій.

Fabric працює за принципом дозволеного (приватного) блокчейну, де учасники спочатку повинні отримати дозвіл на участь у мережі, що підвищує рівень безпеки і контролю над операціями. Це дозволяє організаціям зберігати повний контроль над даними, що є важливим у фінансових установах, державних структурах і в будь-якій сфері, де критично важлива конфіденційність і дотримання нормативних вимог.

Наступною є EOS – це публічна блокчейн-платформа, яка має на меті вирішення ключових проблем, пов’язаних із масштабованістю та продуктивністю блокчейнів. Платформа орієнтована на високі показники

продуктивності, що досягаються за допомогою використання механізму делегованого підтвердження частки (DPoS). Ця модель дозволяє обробляти тисячі транзакцій за секунду, що значно перевищує можливості традиційних блокчейнів, таких як Bitcoin або Ethereum. EOS призначена для масштабних додатків, таких як децентралізовані соціальні мережі, ігри, платіжні системи та інші ресурсоємні децентралізовані сервіси, які потребують високої пропускної здатності мережі.

Основою механізму DPoS є те, що власники токенів EOS обирають делегатів, які відповідають за створення блоків і підтвердження транзакцій. Процес делегованого підтвердження частки DPOS в блокчейн-платформі розглянемо на рис.1.3.

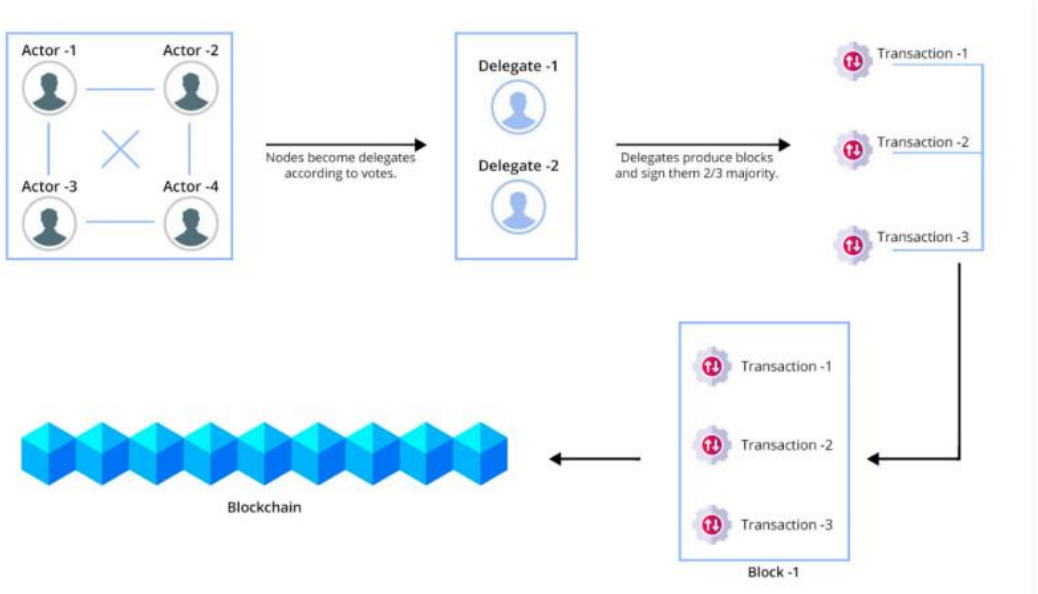

Рис.1.3 Процес делегованого підтвердження частки DPOS в блокчейн-платформі

На зображенні можемо побачити, що на початку в мережі є кілька учасників (Actors), кожен з яких має право голосувати за делегатів. На основі голосування вузли мережі стають делегатами (Delegates), які отримують право на створення блоків. Делегати потім створюють нові блоки, додаючи до них транзакції. Щоб блок був затверджений, його має підписати 2/3 більшості делегатів.

Після того, як транзакції зібрані та оброблені, вони групуються у блок (Block), який додається до існуючого ланцюжка блоків (Blockchain). Таким чином, модель DPoS дозволяє вузлам-делегатам виконувати транзакції та створювати блоки на основі голосування інших учасників, що забезпечує ефективність процесу і швидке підтвердження транзакцій.

Це дозволяє суттєво скоротити час підтвердження транзакцій і підвищити ефективність роботи мережі. На відміну від моделі Proof of Work (PoW), де кожен учасник змагається за право підтвердити блок шляхом виконання складних обчислювальних завдань, у DPoS лише невелика група обраних делегатів має право на створення блоків. Це дозволяє уникнути надмірного споживання ресурсів і сприяє енергетичній ефективності платформи. Крім того, EOS може обробляти транзакції паралельно, завдяки чому забезпечується значне підвищення швидкості роботи мережі.

Смарт-контракти на EOS можна розробляти з використанням мов C++ та WebAssembly (WASM), що робить платформу гнучкою та зручною для програмістів, які звикли до більш традиційних мов програмування. Висока продуктивність платформи також підкріплюється низькими комісіями за транзакції, оскільки модель EOS передбачає використання ресурсів, таких як CPU, RAM і мережевий трафік, що купуються користувачами, а не прямі комісії за кожну транзакцію. Це робить EOS привабливою для великих проєктів і додатків, яким потрібна масштабованість і ефективність.

Проте одним із основних недоліків системи EOS є можлива централізація через модель DPoS. Оскільки невелика кількість делегатів отримує право на створення блоків, це може призвести до концентрації влади серед найбільш активних або заможних учасників мережі, що знижує рівень децентралізації порівняно з іншими блокчейнами, такими як Ethereum чи Bitcoin. І хоча делегати регулярно обираються через голосування власників токенів, існує ризик, що великі власники токенів зможуть впливати на вибори делегатів, що підвищує ймовірність централізації управління мережею.

Tezos — це децентралізована блокчейн-платформа, що забезпечує створення та управління смарт-контрактами, і вирізняється своїми унікальними можливостями, зокрема механізмом самооновлення. Ця особливість дає платформі змогу змінювати та вдосконалювати протокол без необхідності виконувати "хардфорки" (розділення мережі, як це відбувається, наприклад, у Bitcoin або Ethereum). Самооновлення дозволяє мережі інтегрувати нові функції або виправляти помилки через внутрішнє голосування учасників, що робить платформу гнучкою та стійкою до змін.

Механізм голосування в Tezos є одним із ключових компонентів його управління. Власники токенів Tezos (XTZ) можуть брати участь у прийнятті рішень щодо змін у протоколі. Кожна пропозиція на покращення платформи проходить через етапи голосування, де учасники обирають, підтримати чи відхилити зміни. Це демократична модель управління, яка спрямована на уникнення ситуацій, коли виникає необхідність у форках (розділення мережі на дві окремі гілки). Таким чином, Tezos забезпечує сталість свого блокчейну, дозволяючи спільноті впроваджувати зміни поступово та консенсусно, зберігаючи єдність мережі.

Ще однією важливою характеристикою Tezos є використання алгоритму Proof of Stake (PoS). Це алгоритм консенсусу, який дозволяє учасникам мережі створювати нові блоки та підтверджувати транзакції на основі кількості токенів, що вони тримають і делегують для участі в процесі підтвердження. Такий підхід значно знижує енергоспоживання порівняно з алгоритмом Proof of Work (PoW), що використовується в таких блокчейнах, як Bitcoin та Ethereum. У Tezos учасники отримують винагороду за підтвердження блоків пропорційно до своїх внесків, що мотивує їх до підтримки безпеки мережі.

Tezos також підтримує делегований Proof of Stake (DPoS), де власники токенів можуть делегувати свої права на підтвердження блоків іншим учасникам без втрати контролю над своїми активами. Це дозволяє навіть невеликим власникам токенів брати участь у підтримці мережі та отримувати частину винагороди. Така система забезпечує масштабованість і енергетичну

ефективність, оскільки потребує менше обчислювальної потужності для забезпечення консенсусу та обробки транзакцій.

Завдяки своїй модульній архітектурі та самооновлюваній природі, Tezos може інтегрувати нові технології та покращення, не створюючи розколів у спільноті та не змушуючи користувачів обирати між різними версіями блокчейну. Це робить платформу надзвичайно привабливою для довгострокових проєктів і децентралізованих додатків (dApps), які потребують стабільної, еволюційної основи. Архітектуру роботи блокчейн-платформи розглянемо на рис. 1.4.

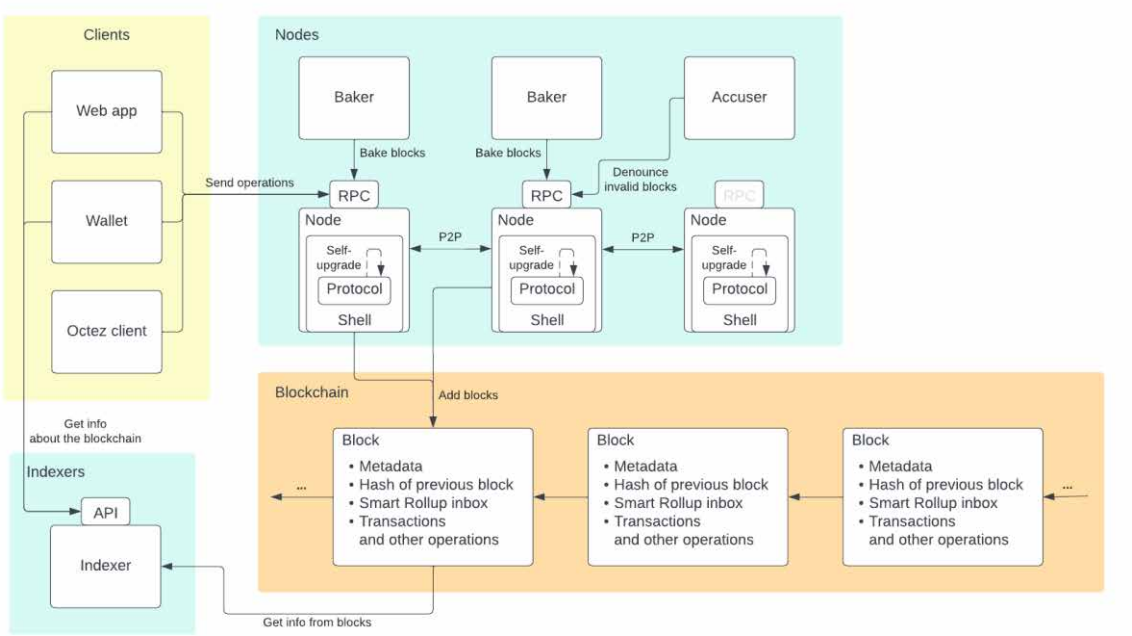

Рис. 1.4 Архітектура блокчейн-платформи Tezos

Клієнти включають веб-додатки, гаманці та клієнт Octez, які взаємодіють із блокчейном через відправлення операцій та отримання інформації про блокчейн за допомогою індексаторів і API. Вузли виконують різні функції: вони можуть створювати нові блоки (процес називається "baking"), відповідати за перевірку валідності блоків та забезпечувати дотримання правил блокчейну. Кожен вузол має протокол та оболонку, які можуть автоматично оновлюватися для підтримки актуальності мережі. Вузли взаємодіють один з одним через механізми P2P для забезпечення синхронізації даних. Блокчейн зберігає всі транзакції та операції у блоках, де кожен блок містить метадані, хеш

попереднього блоку, дані транзакцій та інші операції. Індексатори забезпечують отримання інформації про блоки та надають доступ до неї через API для зовнішніх програм. Економіка Tezos також зосереджена на стимулах для учасників, які підтримують безпеку та ефективність мережі. Оскільки система надає винагороди учасникам за "бейкінг" (створення блоків) і голосування за пропозиції, це створює стійку економічну мотивацію для залучення користувачів до підтримки мережі.

Загалом, Tezos пропонує унікальний підхід до розвитку блокчейну, поєднуючи стійкість до форків, енергоефективність через Proof of Stake та гнучке управління протоколом через механізм голосування. Це робить Tezos однією з найбільш адаптивних та перспективних платформ для розвитку децентралізованих додатків та бізнес-рішень у блокчейн-середовищі. Узагальнюючи весь матеріал розглянемо таблицю 1.2.

Таблиця 1.2

Плюси і мінуси блокчейн-платформ

Платформа	Плюси	Мінуси
Ethereum	- Широка екосистема і велика кількість розробників - Підтримка децентралізованих додатків (dApps) - Популярна мова Solidity	- Високі комісії за транзакції (Gas fees) - Низька масштабованість - Проблеми з продуктивністю під час завантажень
Hyperledger Fabric	- Приватний блокчейн із високою конфіденційністю - Гнучкість налаштування під корпоративні потреби - Підтримка модульної архітектури	- Підходить переважно для приватних систем - Складніша у впровадженні та управлінні в порівнянні з публічними блокчейнами
EOS	- Висока пропускна здатність - Низькі комісії за транзакції - Підтримка C++ та WebAssembly для смарт-контрактів	- Можлива централізація через DPoS - Критика за зниження рівня децентралізації

Tezos	- Самооновлення без необхідності хардфорків - Використання Proof of Stake з низьким енергоспоживанням	- Складніший процес голосування та управління - Порівняно менша спільнота розробників

Аналіз існуючих рішень щодо управління смарт-контрактами показав, що кожна з провідних блокчейн-платформ — Ethereum, Hyperledger Fabric, EOS та Tezos — має свої унікальні переваги та недоліки. Ethereum, як найбільш поширена публічна платформа для смарт-контрактів, пропонує широку екосистему, але стикається з проблемами масштабованості та високими комісіями. Hyperledger Fabric відрізняється високою гнучкістю та приватністю, але підходить більше для корпоративних, ніж для публічних рішень. EOS забезпечує високу пропускну здатність і низькі комісії, проте критикується за можливу централізацію. Tezos вирізняється самооновленням і енергоефективністю завдяки Proof of Stake, але вимагає складнішого процесу управління мережею. Вибір платформи залежить від конкретних потреб проєкту, таких як вимоги до продуктивності, децентралізації та конфіденційності.

1.3 Огляд нормативної бази та вимог до блокчейн-технологій.

Нормативна база для блокчейн-технологій розвивається в умовах швидкої експансії цієї технології у різних секторах економіки. Основними напрямами правового регулювання блокчейну є питання безпеки, конфіденційності, захисту даних, дотримання вимог боротьби з відмиванням грошей (AML), захисту прав споживачів і податкового регулювання.

На міжнародному рівні одним із ключових регуляторів, що займається стандартизацією блокчейн-технологій, є Міжнародна організація зі стандартизації (ISO). У 2017 році був створений технічний комітет ISO/TC 307, який працює над розробкою міжнародних стандартів у галузі блокчейну та розподілених реєстрів [1, с. 5]. Ці стандарти охоплюють управління даними, безпеку, конфіденційність, а також питання сумісності блокчейн-мереж. Одним з найважливіших аспектів є стандартизація смарт-контрактів та методів аутентифікації для забезпечення цілісності транзакцій [2, с. 45].

Європейський Союз активно займається створенням нормативної бази для регулювання блокчейн-технологій у рамках розвитку цифрової економіки. Одним з головних документів є ініціатива Markets in Crypto-Assets (MiCA), яка спрямована на регулювання ринку криптоактивів і створення правових умов для їх використання [3, с. 12]. MiCA визначає правила щодо емітентів криптовалют, зокрема вимоги до прозорості, звітності та захисту інвесторів, що важливо для підтримки довіри до нових технологій. Окрім цього, директива GDPR (General Data Protection Regulation) регулює конфіденційність та обробку персональних даних, що прямо стосується блокчейн-проєктів, які мають враховувати вимоги щодо зберігання та обробки даних користувачів [4, с. 23].

США застосовують різноманітні підходи до регулювання блокчейн-технологій залежно від штату. Наприклад, штат Вайомінг став лідером у впровадженні сприятливого законодавства для блокчейн-компаній. У 2019 році було прийнято низку законів, які визнають криптовалюту як правомірний актив

і регулюють діяльність компаній у сфері блокчейну [5, с. 34]. Водночас на федеральному рівні Securities and Exchange Commission (SEC) та Commodity Futures Trading Commission (CFTC) займаються наглядом за дотриманням законів, пов'язаних з цінними паперами та деривативами у сфері криптовалют [6, с. 57]. Особлива увага приділяється виявленню шахрайства та відмиванню грошей у криптовалютних операціях.

Важливим аспектом регулювання блокчейн-технологій є дотримання вимог боротьби з відмиванням грошей та фінансуванням тероризму (AML/CFT). На міжнародному рівні стандарти встановлюються Фінансовою дією з боротьби з відмиванням грошей (FATF), яка опублікувала керівництво щодо застосування AML до віртуальних активів та провайдерів послуг віртуальних активів [7, с. 63]. Країни-члени FATF повинні запроваджувати ці рекомендації в національне законодавство для забезпечення прозорості криптовалютних операцій та запобігання використанню блокчейну для нелегальних цілей.

Податкове регулювання криптовалют також є ключовим аспектом. У різних юрисдикціях криптовалюта може розглядатися як актив, товар або валюта, що впливає на оподаткування. Наприклад, у США Служба внутрішніх доходів (IRS) класифікує криптовалюту як власність, що означає, що вона підлягає оподаткуванню приросту капіталу при продажу або обміні [8, с. 72]. Європейський Союз також працює над створенням єдиного податкового регулювання для криптовалютних операцій [9, с. 35].

Захист даних та кібербезпека є критичними аспектами для впровадження блокчейн-технологій. Для цього використовуються міжнародні стандарти, такі як ISO/IEC 27001, який регулює управління інформаційною безпекою. Ці стандарти визначають підходи до захисту від кібератак, забезпечення цілісності даних і конфіденційності транзакцій [10, с. 81]. Також важливу роль відіграють національні закони, спрямовані на захист критичної інформаційної інфраструктури, зокрема в таких галузях, як фінанси, охорона здоров'я та державне управління.

Україна активно розвиває правове поле для регулювання блокчейн-технологій та криптовалют, прагнучи забезпечити правовий захист учасників ринку, запобігати фінансовим злочинам та сприяти інтеграції сучасних технологій у державні та бізнес-процеси. Проте законодавча база поки що перебуває на стадії становлення, хоча є декілька важливих нормативних документів, що стосуються блокчейн-технологій та криптовалют.

Закон України "Про віртуальні активи". У вересні 2021 року Верховна Рада України прийняла закон "Про віртуальні активи", який став першим нормативно-правовим актом, що регулює правовий статус віртуальних активів та їх обіг на території України. Закон визначає права та обов’язки учасників ринку віртуальних активів, включно з їх емісарами, посередниками та користувачами [1, с. 23]. Окрім цього, він встановлює правила для платформ, які надають послуги у сфері віртуальних активів, а також визначає умови для запобігання шахрайству та нелегальним операціям з криптовалютами.

Регулювання Національного банку України (НБУ). Національний банк України також бере участь у регулюванні блокчейн-технологій та криптовалют. У 2020 році НБУ оприлюднив аналітичний звіт, у якому детально описані можливі варіанти використання блокчейн-технологій для фінансової системи України. Важливо, що НБУ визнав потенціал технології блокчейн у фінансових транзакціях і розглядає можливість впровадження електронної гривні (e-hryvnia) на базі блокчейн-технологій [2, с. 45]. Проте наразі криптовалюта не має статусу офіційного платіжного засобу в Україні, і всі операції з нею здійснюються на власний ризик користувачів.

Законодавство у сфері фінансового моніторингу. Україна є активним учасником боротьби з відмиванням грошей та фінансуванням тероризму (AML/CFT). У цьому контексті важливу роль відіграє Закон України "Про запобігання та протидію легалізації (відмиванню) доходів, одержаних злочинним шляхом" [3, с. 58]. Закон зобов'язує суб'єктів фінансового моніторингу (включно з криптовалютними біржами та провайдерами гаманців) ідентифікувати своїх клієнтів та повідомляти про підозрілі транзакції. Цей

документ узгоджений із міжнародними стандартами FATF і передбачає посилений контроль за операціями з віртуальними активами для запобігання нелегальній діяльності.

Концепція розвитку цифрової економіки та суспільства України на 2018-2020 роки. Ця концепція була затверджена Кабінетом Міністрів України у 2018 році і передбачає розвиток інноваційних технологій, таких як блокчейн, для покращення державних послуг та прозорості управління. Згідно з документом, Україна планує інтегрувати блокчейн у такі сфери, як реєстрація прав власності, державні закупівлі, та голосування [4, с. 15]. Хоча багато проєктів ще перебувають на стадії тестування, блокчейн-технології визнані ключовими для подальшої цифрової трансформації країни.

Регулювання криптовалютних бірж та обміну віртуальними активами. У рамках Закону "Про віртуальні активи" запроваджено механізми ліцензування криптовалютних бірж та обмінників. Міністерство цифрової трансформації України визначене як регулятор, що відповідатиме за моніторинг і нагляд за діяльністю суб’єктів ринку віртуальних активів. Це дозволяє легалізувати діяльність бірж та обмінників, створюючи захищене середовище для операцій з віртуальними активами та забезпечуючи прозорість процесів [5, с. 37].

Наступним кроком у системному аналізу ПЗ є аналіз і визачення функціональних, нефункціональних вимог.

Функціональні вимоги для системи управління смарт-контрактами, розробленої з використанням технологій блокчейн, описують необхідний функціонал для ефективної обробки смарт-контрактів у середовищі, що емуляє блокчейн. Основні компоненти вимог охоплюють можливості для емулювання блокчейну, створення та управління смарт-контрактами, обробки ставок і аукціонів, підтримки продажу та передачі NFT, а також забезпечення механізмів лотереї і верифікації повідомлень для підвищення безпеки. Крім того, передбачені функції для управління адресами користувачів та контрактів і відстеження дій для подальшого аудиту, що забезпечує повну прозорість у роботі з контрактами. Детальніше розглянемо функціональні вимоги у таблиці 1.3.

Емуляція блокчейну.

- емуляція блоків – забезпечення послідовного створення блоків, кожен з яких містить набір транзакцій, імітуючи структуру реального блокчейну;
- обробка транзакцій – обробка вхідних транзакцій із збереженням повідомлень і метаданих для подальшої роботи контрактів;
- генерація нових блоків – автоматичне створення нових блоків після заповнення поточного блоку для безперервної роботи ланцюга;

Створення і управління смарт-контрактами;

- реєстрація контрактів – запуск нових контрактів із комісією за активацію, з урахуванням початкового балансу та налаштувань методів;
- взаємодія з контрактами – підтримка прямих викликів методів контрактів або взаємодія через транзакції з передачами повідомлень;
- управління транзакціями контрактів – контроль вхідних і вихідних транзакцій контракту, з додатковим хешуванням для безпеки;
- стан контракту – підтримка “сплячого” стану, що відкладає обробку транзакцій до певного часу або події.

Обробка ставок та аукціонів.

- механізм аукціону – підтримка класичних і голландських аукціонів з мінімальними ставками; ставки приймаються до завершення тайм-ауту або досягнення мінімальної ціни;
- ставки на продаж – прийом ставок на продаж токенів (навіть якщо вони не виставлені на аукціон), які власник може схвалити або відхилити.

Підтримка NFT та продажів.

- передача прав власності – можливість передавати права власності на NFT іншим користувачам через транзакції;
- роялті та комісії – автоматичне нарахування роялті творцю та комісії платформі під час продажу або передачі NFT;
- аукціон і продаж NFT – можливість виставлення NFT на продаж або аукціон з установленням мінімальної ціни.

Лотереї та азартні ігри.

- механізм лотереї – підтримка лотерей, що базуються на хешах блоків для визначення результатів і автоматичної виплати виграшів;
- розрахунок виграшів – нарахування виграшів, зокрема множників залежно від кількості виграних номерів або ставок.

Верифікація повідомлень і безпека.

- хешування повідомлень – використання SHA256 для верифікації повідомлень і забезпечення їхньої цілісності;
- захист повідомлень – підтримка додаткової перевірки повідомлень за допомогою 192-бітного хешу.

Управління адресами та обліковими записами.

- облікові записи – надання ідентифікатора для кожного облікового запису з балансом і можливістю здійснення транзакцій;
- адреси контрактів – підтримка адрес, прив'язаних до контрактів, з можливістю переведення в “сплячий” стан.

Логування та відстеження дій.

- журнал подій – реєстрація дій для подальшого аудиту, включаючи створення блоків, транзакцій, контрактів, аукціонів тощо;
- Відправка повідомлень – можливість відправки контрактами повідомлень іншим користувачам або контрактам для відстеження активності.

Ці функціональні вимоги охоплюють усі ключові аспекти системи, необхідні для забезпечення надійної та прозорої роботи децентралізованих смарт-контрактів на платформі блокчейн.

Нефункціональні вимоги є фундаментальними для забезпечення стабільності, ефективності та надійності системи управління смарт-контрактами на блокчейні. Ці вимоги окреслюють не лише технічні параметри, а й загальну якість взаємодії користувачів із системою, охоплюючи характеристики, які забезпечують безпечне, надійне та продуктивне виконання смарт-контрактів.

Нефункціональні вимоги охоплюють такі критично важливі аспекти, як продуктивність — здатність системи обробляти значну кількість запитів і

транзакцій без затримок; безпеку — гарантоване збереження даних і захист від несанкціонованого доступу через криптографічні алгоритми та механізми аутентифікації; і масштабованість — можливість системи адаптуватися до зростаючого обсягу даних, транзакцій і користувачів, забезпечуючи гнучкість і збереження продуктивності. Детальніше нефункціональні вимоги розглянемо детальніше у таблиці 1.3.

Таблиця 1.3

Нефункціональні вимоги до системи

Категорія	Вимоги
Продуктивність	Обробка великої кількості транзакцій та запитів із мінімальною затримкою; швидке підтвердження операцій та низька латентність.
Безпека	Захист від несанкціонованого доступу та маніпуляцій через криптографічні алгоритми, шифрування даних та механізми аутентифікації.
Масштабованість	Підтримка зростаючого обсягу транзакцій, користувачів та даних без втрати продуктивності; можливість горизонтального масштабування.
Зручність використання	Інтуїтивний та простий інтерфейс для налаштування та управління контрактами; документація та навчальні матеріали для користувачів.
Доступність	Постійний доступ до системи 24/7 з мінімальним простоєм; швидке відновлення після збоїв або пікових навантажень.
Сумісність	Підтримка інтеграції з іншими платформами та середовищами; відповідність стандартам блокчейн-мереж для забезпечення інтероперабельності.
Відмовостійкість	Підтримка стабільної роботи при збої одного або кількох компонентів; автоматичне відновлення з мінімальним впливом на роботу системи.
Підтримка та оновлення	Можливість внесення оновлень і поліпшень без порушення роботи поточних контрактів; надання оновлень без значних простоїв системи.

Таким чином, нефункціональні вимоги служать основою для створення надійної, масштабованої, безпечної та продуктивної платформи, що здатна підтримувати динамічний розвиток у сфері управління смарт-контрактами на блокчейні.

1.4 Постановка завдання для розробки системи управління смарт-контрактами.

Метою розробки системи управління смарт-контрактами є створення платформи, що дозволить користувачам ефективно створювати, запускати та керувати смарт-контрактами на основі технології блокчейн. Основне завдання полягає у забезпеченні прозорого та автоматизованого виконання контрактів без необхідності посередників, з можливістю масштабування, високим рівнем безпеки та зручним інтерфейсом для кінцевих користувачів.

1. Розробка інтерфейсу для користувачів

створення зручного веб-інтерфейсу для роботи зі смарт-контрактами, який дозволить користувачам легко створювати, редагувати та запускати контракти, з інтуїтивно зрозумілою навігацією.

2. Механізм управління смарт-контрактами

реалізація функцій розгортання, активації, редагування та завершення смарт-контрактів з автоматичним виконанням умов за алгоритмом підтвердження.

3. Підтримка багатомовності для смарт-контрактів

забезпечення можливості розробки смарт-контрактів мовами програмування, такими як Solidity та C++.

4. Проведення транзакцій

інтеграція фінансових операцій з підтримкою криптовалют і токенів та можливістю підключення до гаманців і обмінників.

5. Масштабованість системи

підтримка великої кількості користувачів і транзакцій без втрати продуктивності за допомогою рішень Layer 2.

6. Безпека та конфіденційність

інтеграція шифрування даних, багаторівневої аутентифікації, захисту від DDoS-атак, резервного копіювання та відповідності політикам AML/CFT.

7. Журнал транзакцій та моніторинг

створення журналу для перегляду історії операцій, що забезпечить прозорість та контроль за виконанням смарт-контрактів.

8. Підтримка мультипідпису

впровадження мультипідпису для критично важливих транзакцій, які потребують підтвердження кількома учасниками для додаткового захисту.

Вимоги до продуктивності та безпеки:

- продуктивність: обробка великої кількості транзакцій за секунду без затримок;
- безпека: захист від атак, шифрування даних, надійна аутентифікація, захист від DDoS;
- Масштабованість: система ефективно працює при збільшенні кількості користувачів та обсягу даних.

Таким чином, система має забезпечувати високий рівень надійності, продуктивності та безпеки, зберігаючи при цьому зручність у використанні для кінцевих користувачів.

2 МОДЕЛЮВАННЯ СИСТЕМИ УПРАВЛІННЯ СМАРТ-КОНТРАКТАМИ

2.1 Вибір підходів до моделювання: функціональний та об'єктно-орієнтований підходи.

При моделюванні системи за допомогою функціонального та об’єктно-орієнтованого підходів у UML використовуються різні типи діаграм, що відображають різні аспекти системи: функціональність, структуру та динаміку її роботи. Вибір підходу залежить від завдань, які необхідно вирішити в процесі моделювання.

Функціональний підхід до моделювання систем полягає у визначенні та описі функціональності системи, зокрема її основних процесів та взаємодії користувачів з цими процесами. Головна мета функціонального підходу – ідентифікувати ключові функції та операції, які виконує система, незалежно від технічної реалізації, з акцентом на те, "що" система робить, а не "як" це реалізовано [10, с. 54]. У процесі моделювання функціональних вимог використовуються діаграми, такі як діаграми прецедентів та активності, які дозволяють зрозуміти бізнес-процеси та логіку роботи системи на високому рівні абстракції.

Функціональний підхід надає можливість аналізувати взаємодію між користувачами та системою, забезпечуючи зрозуміле представлення функціональних можливостей системи для всіх учасників проєкту, включаючи кінцевих користувачів, розробників і аналітиків. Таке моделювання дозволяє ефективно визначити вимоги до системи на початкових етапах розробки, що допомагає уникнути багатьох проблем на подальших етапах проєктування та реалізації [10, с. 102].

Діаграма прецедентів (use case diagram) є одним із основних інструментів для функціонального моделювання системи, оскільки вона відображає всі

головні функції, або прецеденти, які система повинна виконувати, а також визначає взаємодію між користувачами (акторами) і самою системою. Основна мета цієї діаграми – надати візуальне уявлення про те, які саме функції має забезпечувати система, і яким чином різні актори будуть задіяні у виконанні цих функцій. Це дозволяє визначити всі функціональні можливості системи та описати головні сценарії її використання, що є важливим як для бізнес-аналітиків, так і для розробників [11, с. 54].

Кожен прецедент, що позначений на діаграмі, представляє конкретну функцію системи, яка має виконувати певне завдання або забезпечувати певну послугу для користувача або іншої системи [11, с. 42]. Наприклад, у системі управління контрактами, прецедентом може бути «Реєстрація нового контракту», а актором — адміністратор, який відповідає за створення та управління контрактами [11, с. 102]. Завдяки такій візуалізації, діаграма прецедентів полегшує комунікацію між учасниками проєкту та дозволяє зацікавленим сторонам чітко розуміти, які саме функції буде виконувати система і яким чином вони зможуть нею користуватися [11, с. 17]. Розглянемо діаграму прецедентів для нашої системи детальніше на рис.2.1.

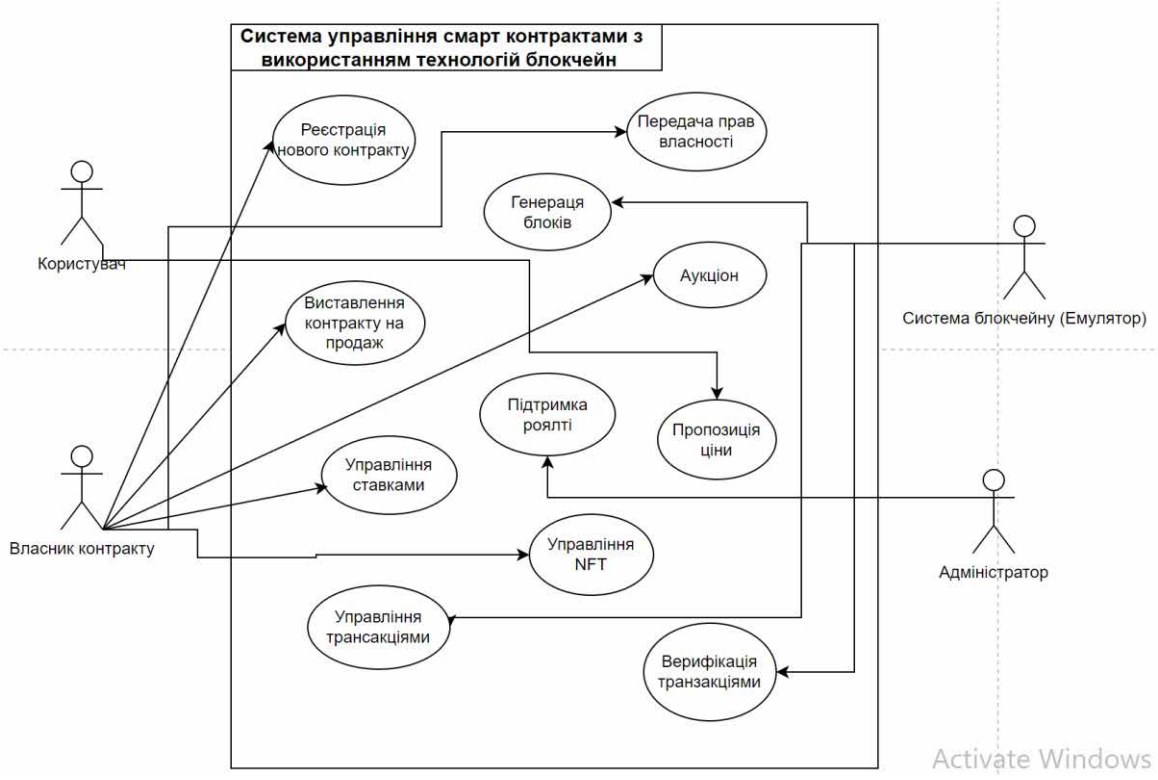

Рис. 2.1 Діаграма прецедентів системи управління смарт контрактами

Розглянемо детальніше діаграму прецедентів, головними акторами системи є «Користувач», «Система блокчейну», «Власник контракту», «Адміністратор», і відповідно ці користувачі взаємодіють з системою (за допомогою різноманітного функціоналу). Весь функціонал у системі є зрозумілим, єдине що функція верифікації транкзакцій реалізована нами за допомогою хешування SHA-256, що може гаранувати нам основне – цілініість і безпеку даних, що і є основною метою технології, а саме захист потоків даних. Узагальнюючи, система надає дуже великий функціонал для користувача.

Розглянемо далі діаграма послідовності, яка є одной з основних UML діаграм, вона відображає послідовність дій або кроків, що можуть виконуватись у нашій системі. Діаграма показує як обробляються події і виконуються дії у вигляді послідовності кроків, що може включати розгалуження, паралельне виконання, цикли та кінцеві стани. Відповідно основною метою діаграми активності є візуалізація процесів, це допомагає зрозуміти алгоритм роботи системи або конкретного процесу, що є дуже корисно для аналізу та оптимізації бізнес-процесів або для деталізації логіки роботи програмного забезпечення. Розглянемо діаграму активності детальніше на рис. 2.2.

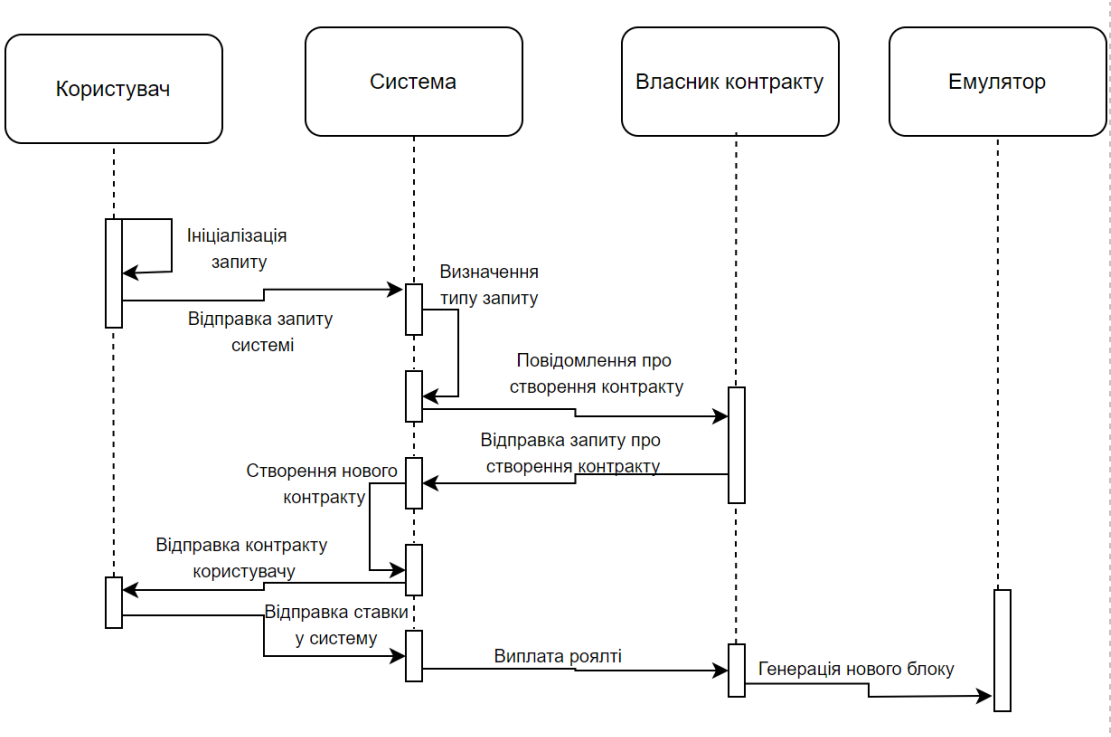

Рис.2.2 Діаграма послідовності системи

«Користувач» ініціює запит до «Системи», яке може стосуватись до створення нового контракту або виконання іншої дії. Відповідно після отримання запиту, «Система» аналізує його і визначає тип необхідної операції. Якщо запит стосується створення нового контракту, «Система» надсилає повідомлення «Власнику контракту» для підтвердження, що контракт буде створено.

Після цього «Власник контракту», отримавши повідомлення від «Системи», підтверджує створення контракту, відправляючи відповідний запит. Далі «Система» створює новий контракт і надсилає його «Користувачу», який тепер може взаємодіяти з ним або використовувати його у подальших транзакціях. Якщо «Користувач» здійснює ставку або оплату в межах контракту, він надсилає цю інформацію до «Системи» яка обробляє отриману ставку і, якщо це передбачено умовами, виплачує роялті «Власнику контракту».

Завершальним процесом є дії емулятора, «Емулятор» виконує генерацію нового блоку, який фіксує всі зміни, транзакції та оновлення, пов’язані із запитом «Користувача», а саме Генерація нового блоку гарантує, що дані залишаються цілісними та захищеними в рамках блокчейну.

Діаграма активності, що представлена на рис. 2.3, нагадує блок-схему, яка графічно описує алгоритм розв’язку задачі. В ній ми бачимо представлення станів системи. Кожен стан відповідає виконанню певної операції і перейти до наступного стану можливо лише по завершенню операції в початковому стані.

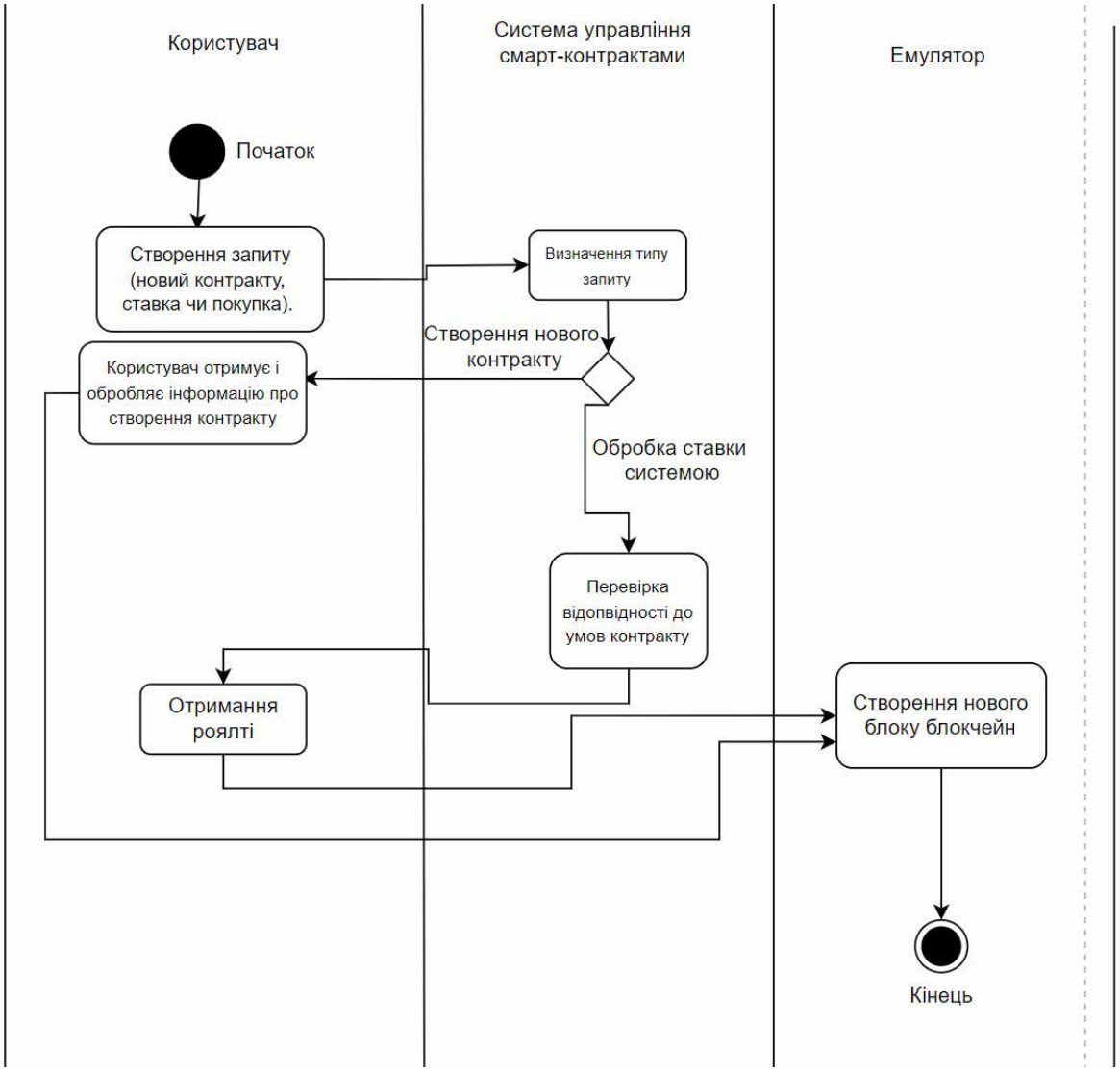

Рис. 2.3 Діаграма активності системи

Спочатку «Користувач» ініціює запит на виконання певної дії, такої як створення нового контракту, здійснення ставки або покупка. Цей запит передається в систему управління смарт-контрактами, яка визначає його тип.

Якщо запит стосується створення нового контракту, система переходить до процесу створення контракту, після чого "Користувач" отримує та обробляє інформацію про успішне створення контракту. У випадку, коли запит пов'язаний із ставкою або покупкою, система переходить до обробки ставки. В рамках цього процесу система перевіряє відповідність ставки або покупки до умов контракту.

Після перевірки та виконання необхідних дій, "Користувач" отримує роялті, якщо це передбачено умовами контракту. Усі завершені дії фіксуються у новому блоці блокчейну, який генерується емулюючою системою

("Емулятором"). Завершується процес у системі після створення нового блоку, що зберігає всі зміни та забезпечує цілісність даних у блокчейн-системі.

Об'єктно-орієнтований підхід концентрується на структурі системи через поняття об'єктів, які репрезентують окремі елементи з їхніми даними та функціональністю. Кожен об'єкт є автономною сутністю, що об'єднує в собі як стан (властивості), так і поведінку (методи), що робить його віддзеркаленням реальних речей або абстрактних концепцій. Така модель дозволяє системі відображати складніші, взаємопов'язані структури, зосереджуючи увагу на тому, як об'єкти взаємодіють один з одним через методи та зв’язки.

Завдяки об'єктно-орієнтованому підходу можна створювати гнучкі й модульні системи, які легко масштабуються і змінюються відповідно до потреб. У цьому підході важливими є зв'язки між об'єктами, які можуть включати асоціацію, агрегацію, композицію та наслідування. Наслідування дозволяє новим класам успадковувати характеристики від базових класів, що знижує дублювання коду та підвищує його повторне використання. Композиція та агрегація описують, як об’єкти можуть поєднуватися, створюючи нові, більш складні структури.

Основними принципами об'єктно-орієнтованого підходу є інкапсуляція, наслідування та поліморфізм. Інкапсуляція обмежує доступ до внутрішніх деталей об'єкта, дозволяючи змінювати реалізацію без впливу на зовнішній код. Поліморфізм же дозволяє об'єктам взаємодіяти через загальні інтерфейси, але виконувати дії по-різному залежно від конкретної реалізації.

Діаграма класів — це тип статичної структурної діаграми в мові уніфікованого моделювання (UML), що представляє схему системи, зображаючи її класи, атрибути, операції (або методи) та зв’язки між цими класами. Слугує як візуальне уявлення структури системи, діаграма класів допомагає зрозуміти взаємодії, залежності та ієрархію різних компонентів системи.

Кожен клас на діаграмі зображений як прямокутник, розділений на три частини. У верхній частині зазначено ім'я класу, в середній — атрибути класу (такі як змінні або властивості), а в нижній містяться методи або операції, які

може виконувати клас. Класи часто з'єднуються лініями, що показують такі зв'язки, як наслідування, асоціація або залежність.

- наслідування: зображено суцільною лінією зі стрілкою, спрямованою від похідного класу до базового класу, що означає, що один клас наслідує властивості та поведінку іншого;
- асоціація: проста лінія, що з'єднує два класи, часто з маркерами кратності, показує зв'язок між класами, де один клас використовує або залежить від іншого;
- залежність: пунктирна лінія зі стрілкою представляє слабкий зв'язок, коли один клас залежить від іншого для належного функціонування, але ця залежність не є фундаментальною;
- Агрегація та композиція: агрегація зображується відкритим ромбом на кінці зв'язку, позначаючи відношення "ціле-частина". Композиція, яка є сильнішою формою агрегації, позначається заповненим ромбом, вказуючи, що "частина" не може існувати незалежно від "цілого".

Діаграми класів забезпечують чітке уявлення про статичні зв’язки в системі, підтримуючи ефективне проектування, розробку та комунікацію між зацікавленими сторонами. Вони особливо корисні для об'єктно-орієнтованого програмування, оскільки пропонують візуальне розбиття того, як класи взаємодіють, що забезпечує організований і легкодоступний для підтримки код.

Розглянемо діаграму класів детальніше на рис. 2.4.

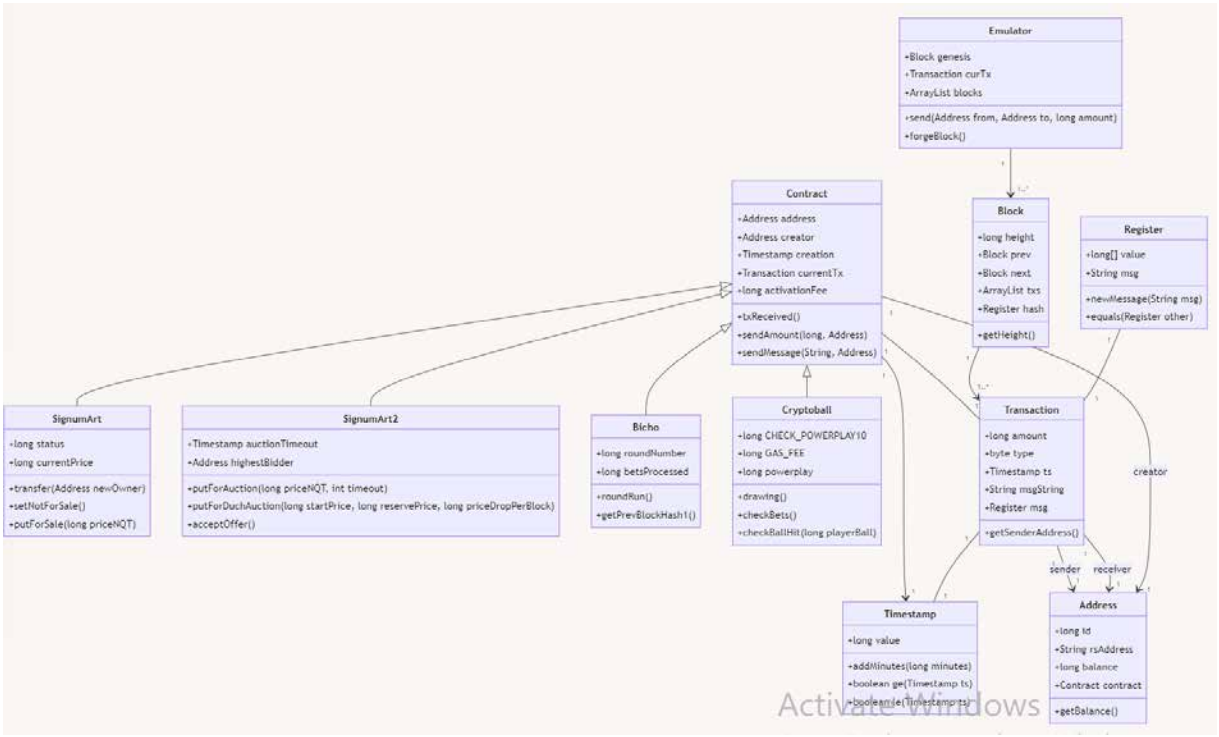

Рис. 2.4 Діаграма класів програмної системи

Основний клас, Contract, є базовим для всіх смарт-контрактів у системі. Він містить атрибути, як-от адреса контракту, адреса творця, час створення, поточна трансакція, та активаційний платіж, а також методи для надсилання трансакцій і повідомлень. Класи, такі як SignumArt, SignumArt2, Bicho та Cryptoball, успадковують функціонал Contract і розширюють його специфічними можливостями.

Клас SignumArt представляє смарт-контракти для NFT або цифрових активів із функціями продажу та передачі прав власності, тоді як SignumArt2 розширює ці можливості, додаючи механізми для звичайного та голландського аукціонів. Клас Bicho реалізує гру або лотерею зі зберіганням номера раунду та кількості ставок, а також має методи для управління ігровими раундами. Cryptoball, ще один клас для лотереї, зберігає та обробляє ставки, генерує випадкові числа та визначає виграшні результати.

Emulator відповідає за емуляцію блокчейн-функцій, створює блоки та обробляє трансакції, а також взаємодіє з блоками та адресами в системі. Клас Block зберігає основну інформацію про блок, включаючи його висоту, хеш та список трансакцій, представляючи структуру, яка моделює блокчейн. Клас

Transaction представляє окрему трансакцію, містить інформацію про відправника, отримувача, суму, тип трансакції та часову мітку, а також пов'язаний з блоком, в якому ця трансакція зберігається.

Клас Register використовується для зберігання повідомлень та хешів, забезпечуючи методи для створення та порівняння хешів. Клас Timestamp представляє часову мітку, яка використовується для контролю часу трансакцій і блоків, з методами для додавання хвилин і порівняння часових міток. Address представляє адресу в системі та містить інформацію про баланс, ідентифікатор, RS-адресу та прив'язку до смарт-контракту, дозволяючи проводити операції та взаємодіяти з іншими класами.

Взаємодія між класами полягає у взаємному обміні інформацією та операціями: Emulator пов’язаний із класами Block та Transaction для керування обробкою блоків та трансакцій. Contract взаємодіє з Address, Transaction та Timestamp, оскільки кожен контракт має адресу, може отримувати трансакції та оперувати з часовими мітками для відстеження подій.

Загалом, діаграма відображає систему, яка забезпечує управління цифровими активами, лотереями, ставками та аукціонами, надаючи структуру для безпечної взаємодії користувачів із контрактами та елементами блокчейну, що емулюються, дозволяючи реалізувати децентралізовану систему управління смарт-контрактами.

2.2 Логічна та фізична модель даних

Логічна модель даних представляється за допомогою ERD (Entity-Relationship diagram). Дана діаграма дозволяє представити модель майбутньої реляційної БД. Дані діаграми допомагають розробникам краще зрозуміти структуру даних, яка використовується у системі та виявляти потребу в нормалізації даних.

При створенні ER діаграми використовуються такі елементи, як атрибути, таблиці та зв'язки, зокрема зв'язки діляться на 2 типи: ідентифікуючий та неідентифікуючий. Ідентифікуючий зв'язок дозволяє включити зовнішній ключ в групу первинного ключа таблиці, через що зв'язок між таблицями стає міцнішим і це гарантує що записи в дочірній таблиці не можуть існувати без батьківської. В свою чергу через неідентифікуючий зв'язок зовнішній ключ не вноситься в групу первинного ключа, тому записи в дочірній таблиці можуть існувати без запису в батьківській.

Узагальному, виділяють три типи деталізації моделей дани, а саме концептуальна модель – найвищий рівень представлення, яка відповідно передбачає мінімальну кількість деталей і показує загальну схему даних. Наступним є логічна модель яка містить більш детальну інформацію, і включає в себе визначення сутностей, атрибутів та зв'язків між ними. І фізична модель (на основі логічної моделі будується фізична модель) містить всі технічні деталі, які необхідні для розробки та реалізації бази даних.

Розглянемо детальніше ER діаграму для нашої системи на рис. 2.5, вона надасть нам структурований огляд зв'язків та взаємозв'язків у блокчейн системі, ці всі сутності відображають необхідні елементи для забезпечення важливої безпечної взаємодії.

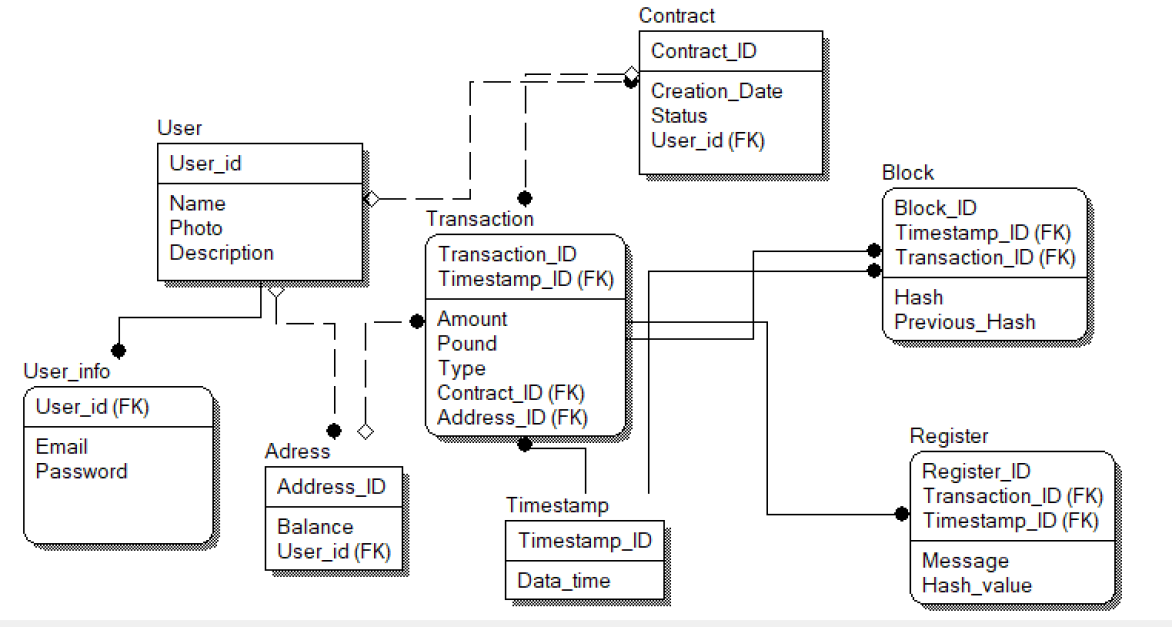

Рис.2.5 ER діаграма у нашій системі

Розглянемо сутності детальніше у таблиці 2.1

Таблиця 2.1

Сутності та їх атрибути

Сутність	Атрибути	Опис
User	User_id (PK), Name, Photo, Description	Ідентифікатор, ім'я, фото та опис.
User_info	User_id (FK), Email, Password	Зберігає додаткову інформацію про користувача, включаючи email і пароль, пов'язана з User.
Contract	Contract_ID (PK), Creation_Date, Status, User_id	Включає дату створення, статус та зв'язок із користувачем User.
Transaction	Transaction_ID (PK), Timestamp_ID (FK), Amount, Pound, Type, Contract_ID (FK),	Відображає окрему транзакцію, включає суму, тип, зв'язки з контрактом та адресою.
Address	Address_ID (PK), Balance, User_id (FK)	Представляє адресу в блокчейні, зберігає баланс і зв'язок із користувачем User.
Timestamp	Timestamp_ID (PK), Data_time	Зберігає інформацію про дату та час події.
Block	Block_ID (PK), Timestamp_ID (FK), Transaction_ID (FK), Hash, Previous_Hash	Блок у блокчейні, включає хеш, попередній, та зв'язки з транзакціями.
Register	Register_ID (PK), Transaction_ID (FK), Timestamp_ID (FK), Message, Hash_value	Зберігає повідомлення та хеш-значення, пов'язані з транзакціями, включає зв'язки з Transaction.

Фізична модель даних відображає конкретну реалізацію логічної моделі в рамках обраної системи керування базами даних (СКБД). Вона описує структуру бази даних, яка визначає, як саме дані зберігаються на фізичному рівні для забезпечення ефективної роботи. Ця модель включає в себе всі технічні деталі зберігання, включаючи типи даних, індекси, ключі та обмеження, необхідні для підтримки цілісності даних і забезпечення продуктивності системи.

Фізична модель починається з визначення таблиць бази даних, що представляють основні сутності системи. Наприклад, у системі управління смарт-контрактами ми можемо мати такі таблиці, як User, Transaction, Contract, Address. Кожна таблиця описує певну сутність з визначеними стовпцями, що відповідають атрибутам цієї сутності. Для кожного атрибуту визначається тип даних. Наприклад, User може мати стовпці user_id (типу INTEGER), name (типу VARCHAR), photo (типу BLOB для зберігання зображень), що забезпечує уніфікований формат зберігання та доступу до цих даних.

Ключовим аспектом фізичної моделі є визначення ключів та обмежень. Кожна таблиця має первинний ключ, що гарантує унікальність записів. Наприклад, user_id в таблиці User або transaction_id в таблиці Transaction. Зовнішні ключі використовуються для встановлення зв'язків між таблицями. Наприклад, у таблиці Transaction зовнішній ключ contract_id може посилатися на Contract для зв’язку трансакцій з контрактами. Такі зв'язки гарантують логічну цілісність даних і дозволяють будувати складні запити, об'єднуючи дані з кількох таблиць.

Індекси в фізичній моделі забезпечують швидкий доступ до часто використовуваних полів, підвищуючи швидкодію запитів. Наприклад, індекси можна додати до полів transaction_id або timestamp у таблиці Transaction, щоб швидше виконувати запити, що часто звертаються до цих полів.

У деяких випадках застосовується нормалізація, щоб уникнути надлишковості даних, або денормалізація, щоб прискорити виконання запитів. У нормалізованій базі даних дані розбиваються на окремі таблиці, щоб уникнути дублювання, наприклад, дані користувача (User_info) можуть зберігатися окремо

від основної таблиці User. Денормалізація може застосовуватися в тих випадках, коли важлива швидкодія, і допускається певний рівень надлишковості даних. Розглянемо фізичну модель даних для нашої системи на рис. 2.6.

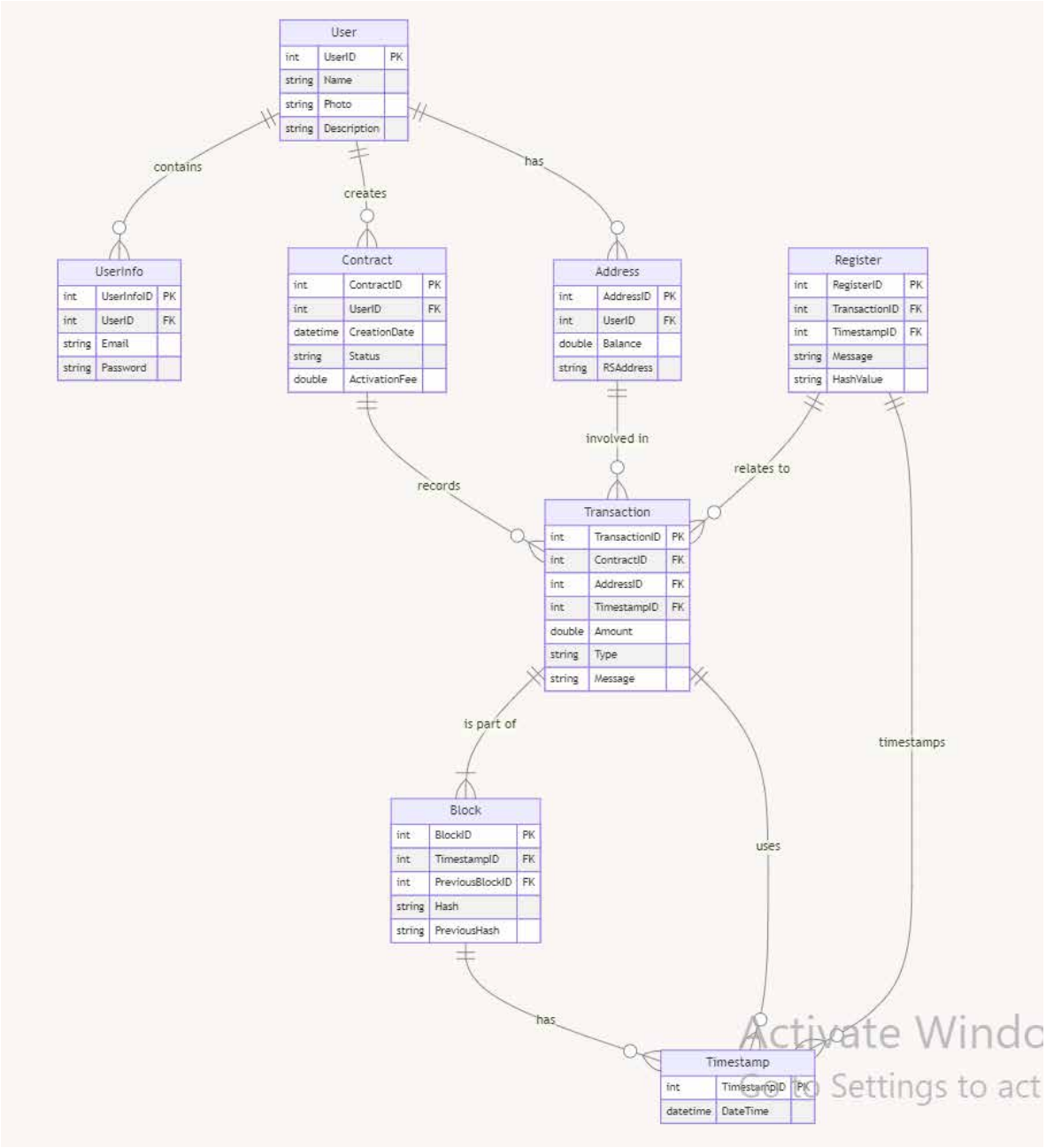

Рис.2.6 Фізична модель даних програмної системи

Таким чином, фізична модель є технічною інструкцією для створення структури бази даних. Вона гарантує, що дані будуть зберігатися ефективно, з можливістю швидкого доступу до них і забезпеченням цілісності. Така модель є важливою основою для побудови системи, яка може працювати з великими обсягами даних та обслуговувати різноманітні запити користувачів.

2.3 Опис архітектури системи на основі блокчейн

Архітектура системи управління смарт-контрактами на основі блокчейну створена для забезпечення надійного і безпечного управління контрактами, їх виконання та збереження даних. Система використовує децентралізовану технологію, що дозволяє забезпечити високу надійність, масштабованість і безпеку. Компоненти системи взаємодіють на різних рівнях, кожен з яких відповідає за окремі функціональні завдання. Розглянемо детальніше архітектуру системи на рис. 2.7.

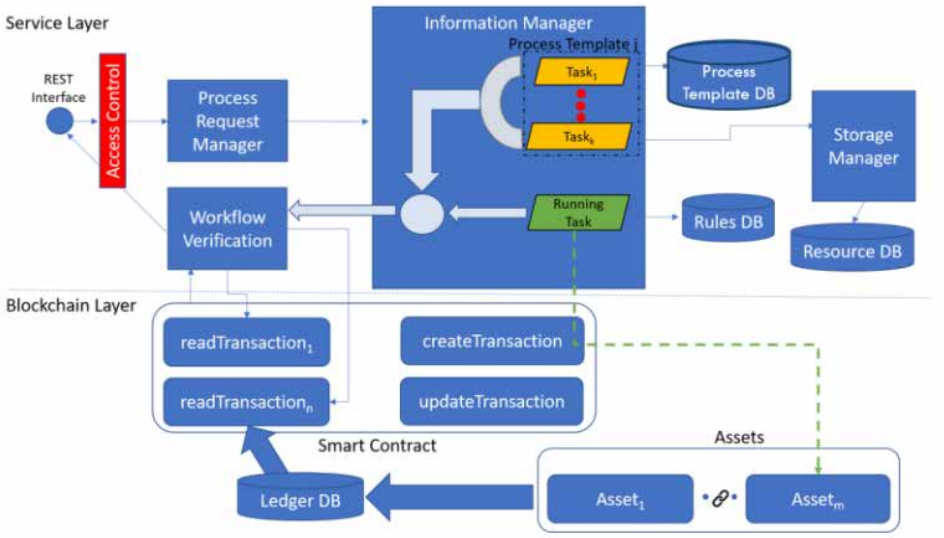

Рис.2.7 Архітектура системи на основі блокчейн

Архітектура складається з сервісного рівня та блокчейн-рівня, які працюють разом для управління, верифікації та зберігання даних у блокчейн-системі. У контексті системи управління смарт-контрактами, що використовує блокчейн, кожен компонент відіграє важливу роль, забезпечуючи безперебійну взаємодію між користувачами та блокчейн-інфраструктурою.

На сервісному рівні запити користувачів обробляються за допомогою менеджера запитів на обробку, який отримує запити через REST-інтерфейс. Контроль доступу забезпечує, що тільки авторизовані користувачі можуть взаємодіяти з системою. Після валідації запити проходять через верифікацію робочого процесу та менеджер інформації. Менеджер інформації відповідає за управління процесами, використовуючи базу даних шаблонів процесів для

завантаження та керування специфічними шаблонами процесів. Ці шаблони визначають структуру та поведінку різних завдань, які може виконувати смарт-контракт. Менеджер зберігання даних підключається до таких баз даних, як база даних правил та база даних ресурсів, які містять попередньо визначені правила та доступні ресурси, необхідні для виконання контракту. Виконувані завдання постійно моніторяться та керуються, забезпечуючи, що кожна дія відповідає вимогам робочого процесу та логіки системи.

На блокчейн-рівні знаходяться основні блокчейн-операції та смарт-контракти. Тут транзакції, пов'язані зі створенням, оновленням та читанням активів, записуються в Ledger DB. Система обробляє такі операції, як readTransaction_1 до readTransaction_n, а також виконує дії на кшталт createTransaction і updateTransaction. Ці транзакції взаємодіють із різними активами, представленими Asset_1 до Asset_m, що можуть означати конкретні екземпляри контрактів, користувачів або інші дані, які відслідковуються на блокчейні. Смарт-контракт взаємодіє з Ledger DB для запису історії транзакцій і змін стану активів.

У цій конфігурації смарт-контракти автоматизують верифікацію та виконання завдань, забезпечуючи цілісність і безпеку даних у блокчейні. Ledger DB зберігає незмінну історію всіх транзакцій, до якої можна отримати доступ і верифікувати за необхідності. Така архітектура гарантує надійну, захищену від підробок систему для управління смарт-контрактами, дозволяючи користувачам безперешкодно взаємодіяти з блокчейн-активами, водночас зберігаючи чіткий запис усіх виконаних операцій. Така структура є ідеальною для застосувань, які потребують високої прозорості, цілісності даних та автоматизованих процесів у блокчейн-середовищі.

Діаграма компонентів ідеально підходить для того, аби моделювати фізичні частини системи. Дана діаграма демонструє загальні залежності між компонентами. В даній діаграмі компонент – це фізично існуюча частина системи, яка забезпечує реалізацію системи а також її функціональну поведінку.

Розглянемо детальніше діаграму компонентів для нашої системи на рис. 2.8.

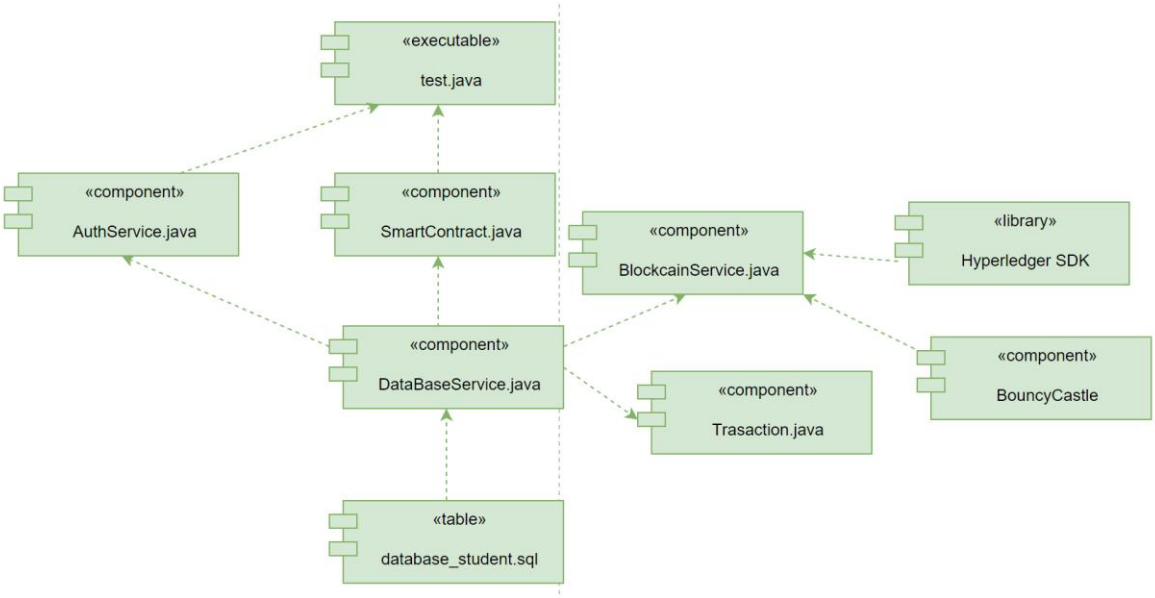

Рис.2.8 Діаграма компонентів нашої системи

Основним компонентом системи є SmartContract.java, який виступає основним обробником логіки для операцій, пов’язаних із блокчейном, виконуючи завдання, такі як запуск контрактів і обробка транзакцій. AuthService.java відповідає за процеси автентифікації, забезпечуючи безпечний доступ для користувачів, які взаємодіють із системою. Цей компонент відповідно використовує криптографічні методи, надані зовнішніми бібліотеками.

DataBaseService.java управляє операціями з базою даних, працюючи з database_student.sql для отримання, зберігання та оновлення відповідної інформації, пов’язаної зі смарт-контрактами чи користувачами. BlockchainService.java виступає посередником між рівнем блокчейну та системою, надаючи можливість смарт-контрактам зчитувати або записувати дані у розподілений реєстр. Для цього, використовується Hyperledger SDK, який забезпечує функціональні можливості для розгортання, виклику та управління смарт-контрактами.

Transaction.java представляє компонент, відповідальний за обробку окремих транзакцій у системі, гарантуючи, що кожна транзакція відповідає

правильному формату та правилам. Крім того, система інтегрує BouncyCastle як криптографічну бібліотеку, забезпечуючи надійні засоби безпеки, необхідні для захищеної обробки транзакцій і даних.

Нарешті, test.java виконує функцію виконуваного компонента, використовуючись для тестування функцій системи, імітуючи реальні сценарії для перевірки цілісності та ефективності процесів, пов'язаних з блокчейном. Зв'язки між компонентами вказують на залежності та взаємодії, які є необхідними для функціонування системи управління смарт-контрактами на основі блокчейну.

У контексті нашого завдання варто розглянути також діаграму пакетів системи. Розглянемо її детальніше на рис. 2.9.

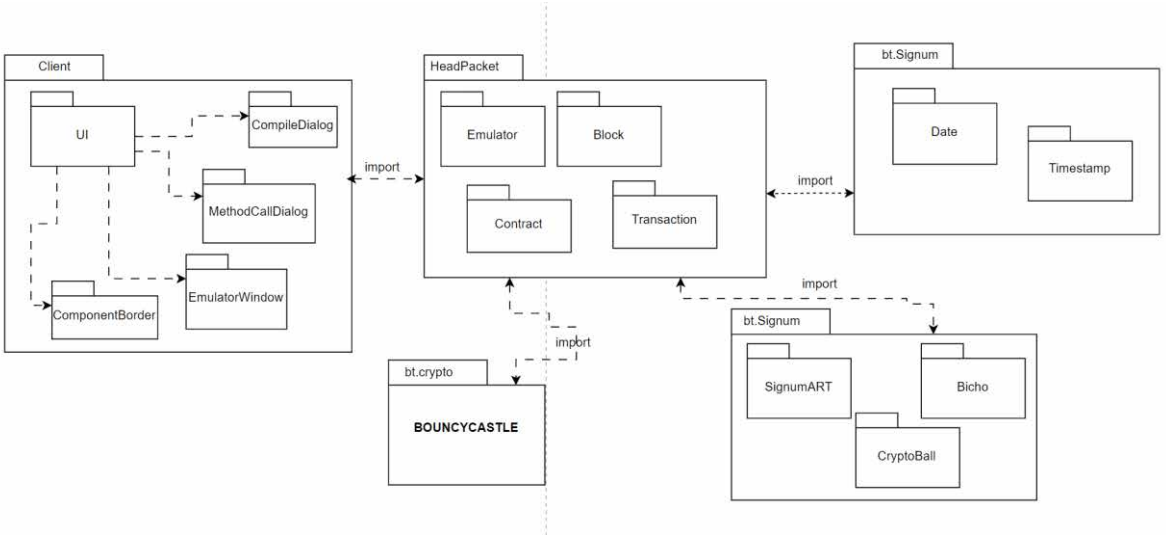

Рис.2.9 Діаграма пакетів програмної системи

Пакет Client представляє інтерфейс користувача (UI) і включає різні компоненти для взаємодії користувача, такі як UI, CompileDialog, MethodCallDialog, ComponentBorder та EmulatorWindow. Ці компоненти забезпечують користувачу можливість працювати з емулятором, а також створювати, компілювати і запускати смарт-контракти.

Пакет HeadPacket виконує роль центрального модуля, що обробляє функціональність блокчейну. Він містить ключові компоненти, такі як Emulator, Block, Contract і Transaction, які відповідають за емуляцію блокчейн-середовища, управління блоками та транзакціями, а також взаємодію з контрактами. Emulator

дозволяє тестувати транзакції та виконання контрактів у безпечному середовищі без необхідності в реальній блокчейн-мережі.

bt.Signum розділений на два підпакети, де перший включає класи Date і Timestamp для управління часовими мітками, що є критично важливим для точного відстеження транзакцій. Другий підпакет містить SignumART і CryptoBall, які представляють конкретні активи на блокчейні, що дозволяє розширити систему для роботи з токенами, NFT та іншими об'єктами цифрової власності.

bt.crypto містить криптографічну бібліотеку BOUNCYCASTLE, яка забезпечує захист даних, що передаються у блокчейні. Цей компонент є важливим для виконання криптографічних функцій, таких як шифрування, цифрові підписи та хешування, що гарантує безпеку та цілісність даних.

Взаємозв'язки між цими пакетами наступні: Client використовує функціональність HeadPacket для доступу до емулятора та взаємодії з блокчейн-системою. HeadPacket покладається на bt.Signum та bt.crypto для управління датами, криптографією та обробкою активів. Загальна структура забезпечує модульність, де кожен компонент виконує окремі завдання, сприяючи простоті обслуговування та масштабованості системи.

2.4 Моделювання сценаріїв використання системи

Розпочнемо з розгляду сценарію коли користувач ініціалізує смарт-контракт при цьому заповнюючи різні параметри. Детальніше можемо побачити на рис. 2.10.

Рис.2.10 Сценарій ініціалізації смарт-контракту

Цей сценарій наочно показує обмежений функціонал системи, у вигляді створення смарт-контракту користувачем.

Далі розглянемо наступний сценарій який відповідає здійсненню транкзакцій у системі. Цей процес є дуже важливим у процесі виконання завдання кваліфікаційної роботи. Детальніше можемо побачити на рис. 2.11.

Рис. 2.11 Сценарій здійснення транкзакцій у системі

Важливо сказати, що моделювання сценаріїв допомагає у написанні коду самої системи, так як наочно можна визначити сутності і їхні взаємозв’язки. При

ініціалізації транкзакції користувачем, система перевіряє доступність коштів після цього записує тракзація у блок і завдяки блокчейн-технології забезпечується прозорість та безпека даних.

Перейдемо до розгляду наступного сценарію, а саме управління NFT, і аналогічно до попередніх розглянемо детальніше на рис. 2.12.

Рис.2.12 Сценарій управління NFT

Користувач може створити і відповідно управляти цифровими активами, потім є можливість висталення на аукціон. Система обробляє цю транкзакцію і проводить збереження кожної зміни власності, що забезпечує прозорість. Ще два сценарії можемо побачити у таблиці 2.1.

Таблиця 2.1

Сценарії використання системи

Сценарій	Опис
Перевірка та аудит	Користувачі мають доступ до перевірки транзакцій, що забезпечує прозорість та контроль, відповідно вони можуть генерувати звіти про свої фінансові операції або статус конкретного контракту, де звіти базуються на даних, збережених у блокчейні.
Тестування контрактів на емуляторі	Перед випуском контракту в реальну блокчейн-мережу, користувачі можуть протестувати його на емуляторі, що дозволяє перевірити логіку виконання контракту та умови без фінансових ризиків, надаючи можливість поліпшити або налаштувати контракт перед реальним застосуванням.

Таблиця надає структурований огляд додаткових сценаріїв у системі, ці всі ключові взаємодіють дають розуміти про багатофункціональність системи.

3 РОЗРОБКА СИСТЕМИ УПРАВЛІННЯ СМАРТ-КОНТАКТАМИ

3.1 Технології та інструменти для реалізації системи

Платформа Signum була обрана для реалізації блокчейн-системи завдяки її унікальним властивостям, які забезпечують високу безпеку, децентралізацію та економічну ефективність смарт-контрактів.

Signum є першою платформою, яка інтегрувала технологію Automated Transactions (AT), що дозволяє створювати самовиконувані смарт-контракти на блокчейні без потреби в постійному ручному втручанні. Завдяки цьому AT-контракти на Signum не вимагають додаткових обчислень або зовнішніх сервісів для виконання, що робить їх економічно вигідними та надійними. Головною перевагою Signum є те що він працює на принципах децентралізації, забезпечуючи прозорість і захист від підробок даних. Система Proof of Capacity (PoC), яку використовує Signum, дозволяє користувачам зберігати криптографічні дані на дисках для забезпечення майнінгу, що знижує енергоспоживання, на відміну від більш традиційних PoW-систем. Спрощену архітектуру принципу проілюстрували на рис. 3.1.

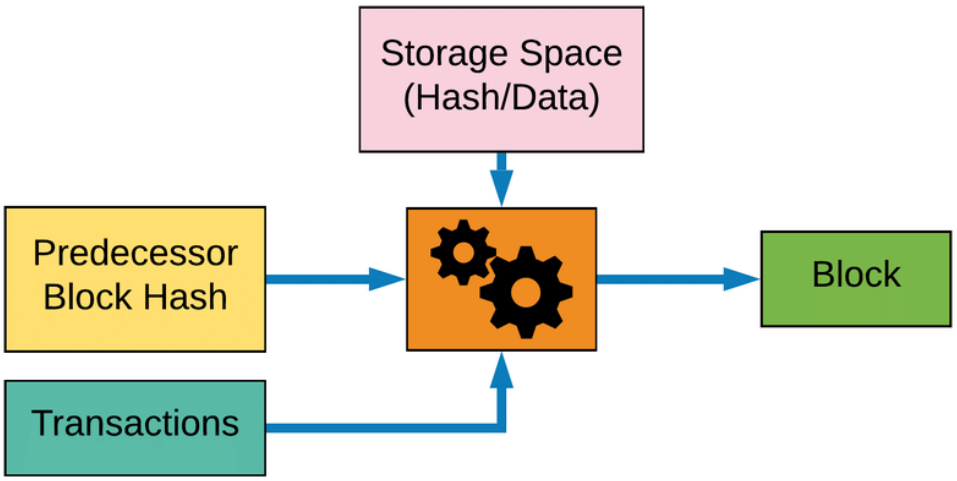

Рис.3.1 Спрощена архітекта PoC

У центрі зображено процес створення блоку, символізований шестернями, які позначають обчислювальні зусилля, необхідні для формування нового блоку в ланцюзі. Відповідно самий процес починається з отримання хеша попереднього блоку, який є криптографічним посиланням на попередній блок у ланцюзі. Цей хеш забезпечує надійне з'єднання кожного блоку з його попередником, що гарантує цілісність та незмінність блокчейну. У процес створення блоку також подаються транзакції, які представляють фактичні дані або записи, що будуть збережені всередині блоку.

Після обробки хеша попереднього блоку та транзакцій результатом стає новий блок. Цей блок включає унікальний хеш, який генерується з транзакцій та хеша попереднього блоку, що потім використовується як посилання для наступного блоку. На діаграмі також показано місце для зберігання зверху, що свідчить про те, що блокчейн зберігає як хеш, так і дані у кожному блоці, забезпечуючи безпеку та надмірність даних у розподіленій книзі.

Розглянемо наступні аспекти обрання платформи Signum, яка є економічно ефективною завдяки PoC. Це означає, що користувачі можуть брати участь у майнінгу без дорогого обладнання та високих витрат на електроенергію, що робить її екологічно дружньою альтернативою іншим блокчейн-платформам. Розглянемо це положення на прикладі наступного графіку на рис.3.2

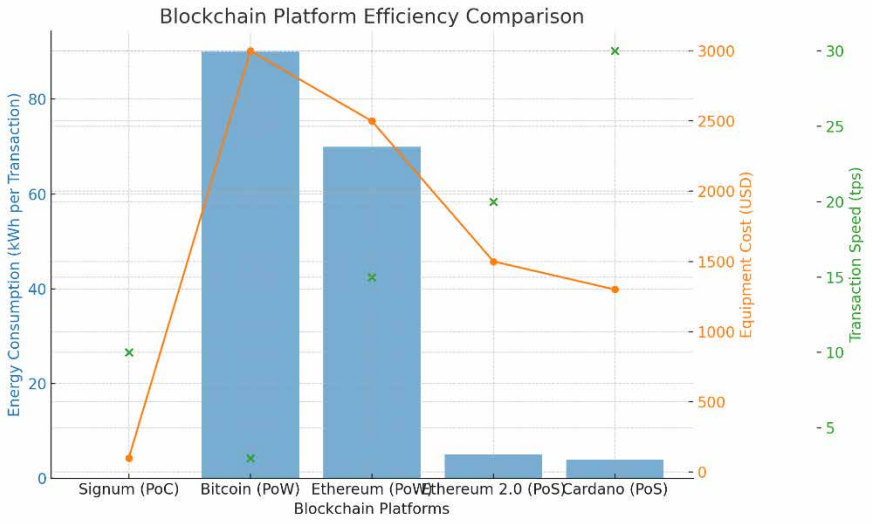

Рис. 3.2 Порівняння ефективності блокчейн-платформ

Signum (на основі алгоритму Proof of Capacity) демонструє найнижче енергоспоживання — лише 0,1 кВт·год на транзакцію, що значно менше, ніж у платформ на основі Proof of Work (Bitcoin та Ethereum), де енергоспоживання сягає 90 і 70 кВт·год відповідно. Вартість обладнання для Signum також нижча, становлячи близько 100 доларів, що значно економніше порівняно з 3000 доларів для Bitcoin. Швидкість транзакцій у Signum (10 транзакцій за секунду) поступається таким платформам, як Cardano, але все ще забезпечує достатню пропускну здатність для багатьох застосувань.

Узагальнюючи, вибір Signum як платформи для даної системи обґрунтований її здатністю підтримувати економічно ефективні та автоматизовані смарт-контракти, що підходить для організації аукціонів та інших фінансових операцій. На відміну від більш традиційних платформ, як-от Ethereum, де транзакційні збори можуть значно зростати, Signum пропонує стабільні та низькі комісії, що особливо важливо для регулярних транзакцій на платформі з високою активністю користувачів.

Для забезпечення сумісності з блокчейн-платформою необхідно використовувати Java 17 або новішу версію. Це дозволяє досягти стабільної інтеграції Java і Gradle, що особливо важливо для роботи з сучасними бібліотеками і технологіями в середовищі блокчейн. Java 17 підтримує нові стандарти безпеки та функціональності, які будуть необхідні для взаємодії з смарт-контрактами та забезпечення стабільності роботи.

Web3j – це легка Java-бібліотека для взаємодії з блокчейном Ethereum. Вона дозволяє легко інтегруватися з смарт-контрактами Ethereum, забезпечуючи можливість відправляти транзакції, отримувати дані з блокчейна та керувати гаманцями без необхідності глибокого знання низькорівневих протоколів. Web3j дає змогу розробникам працювати з Ethereum прямо з Java, що значно спрощує розробку блокчейн-застосунків.

Бібліотека BouncyCastle надає криптографічні API для Java, які є надзвичайно корисними для забезпечення безпеки системи. Завдяки цій бібліотеці можна використовувати алгоритми хешування, шифрування та

перевірки підписів, що особливо важливо під час роботи з конфіденційними даними в транзакціях. BouncyCastle дозволяє захищати дані та забезпечувати їхню цілісність, що є критичним аспектом для будь-якої блокчейн-системи, яка працює з фінансовими або приватними даними.

Gradle є потужним інструментом для управління залежностями проекту, а також для його компіляції та тестування. Використання Gradle полегшує процес розробки, дозволяючи автоматизувати збірку, перевірку залежностей та виконання тестів. Оновлення до Gradle 8.5 гарантує сумісність із Java 17, що дозволяє зосередитися на розробці основного функціоналу без зайвих технічних проблем. Gradle також полегшує інтеграцію сторонніх бібліотек, таких як Web3j і BouncyCastle. Фрагмент діаграми розгортання системи розглянемо на рис.3.3.

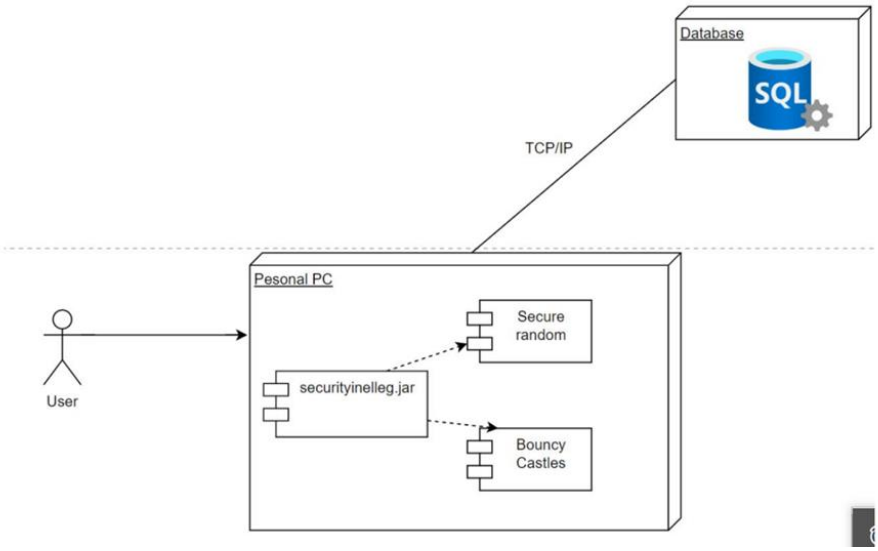

Рис. 3.3 Діаграма розгортання програмної системи

Для тестування і налагодження логіки контрактів у контрольованому середовищі в коді використовуються класи Emulator та EmulatorWindow. Вони дозволяють моделювати блокчейн-процеси без необхідності реального розгортання на тестовій мережі. Завдяки цьому розробники можуть швидко виявляти помилки в логіці контрактів, перевіряти, як вони реагують на різні типи транзакцій, і переконуватися у правильності виконання перед розгортанням у продакшн.

3.2 Криптографічні алгоритми, реалізація алгоритмів смарт контрактів

Основною бібліотекой яка використовувалась у нашій програмі є BouncyCastle для забезпечення криптографічних функцій, таких як хешування і підпис, що є дуже важливими аспектами для захисту транзакцій і валідації даних у блокчейні.

Першочергово розглянемо алгоритм хешування SHA-256 який є основним криптографічним алгоритмом, що використовується для генерації унікального хешу для блоків та транзакцій. Метод performSHA256_64 реалізує SHA-256 хешування, яке генерує хеш із 64-бітного значення. Розглянемо фрагмент реалізації на рис. 3.4.

```
protected long performSHA256_64(long input1, long input2) {
    Register input = new Register();
    input.value[0] = input1;
    input.value[1] = input2;

    Register ret = performSHA256_(input);
    return ret.getValue1();
}
```

Рис. 3.4 Фрагмент реалізації SHA-256

Цей метод бере два довгі числа як вхідні дані, хешує їх і повертає хеш-значення. Метод використовується для забезпечення цілісності даних у блоках та для валідації транзакцій.

Щодо алгоритму PoC, він прямо не реалізований у самому коді. Однак концепція платформи Signum, яка використовує PoC, передбачає, що майнінг і підтвердження транзакцій здійснюються через попереднє збереження даних, які перевіряють майнери під час обробки блоків. PoC в основному використовується на рівні платформи, а не в коді окремого смарт-контракту.

Програма більше орієнтована на реалізацію смарт-контрактів, таких як Bicho, Cryptoball, і SignumArt, які діють у середовищі блокчейн Signum. Смарт-контракти тут використовують функції платформи, такі як транзакції, генерація хешів і симуляція блокчейну для тестування, але вони не вимагають прямого

впровадження алгоритму PoC, оскільки PoC — це механізм консенсусу, що підтримується основним рівнем самої блокчейн-мережі Signum.

До цього ми і перейдемо далі, реалізація смарт-контрактів у нашій системі базується на модульній структурі з класами, що відповідають за виконання різних типів контрактів, таких як SignumArt, Bicho, і Cryptoball. Кожен клас має свої методи для управління логікою контракту, як показано у фрагменті коду нижче на рис. 3.5.

```
public void putForAuction(int priceNQT, int timeout) {
    if (highestBidder == null && owner.equals(this.getCurrentTxSender())) {
        status = STATUS_FOR_AUCTION;
        auctionTimeout = getBlockTimestamp().addMinutes(timeout);
        currentPrice = priceNQT;
        sendMessage(status, owner.getId(), currentPrice,
auctionTimeout.getValue(), tracker);
    }
}
```

Рис. 3.5 Виставлення активу на аукціон

Він встановлює статус контракту на "в аукціоні" і зберігає часовий інтервал для завершення аукціону.

Також реалізація смарт-контракту Bicho включає логіку для обробки ставок, розрахунку виграшів та автоматичного здійснення виплат переможцям. Це дозволяє виконувати транзакції у децентралізованому режимі. Фрагмент коду нижче показує логіку для виплати на рис. 3.6:

```
private void pay() {
    amountToPlatform = currentPrice * platformFee / THOUSAND;
    amountToRoyalties = currentPrice * royaltiesFee / THOUSAND;

    totalPlatformFee += amountToPlatform;
    totalRoyaltiesFee += amountToRoyalties;
    totalTimesSold++;

    sendAmount(amountToRoyalties, royaltiesOwner);
    sendAmount(currentPrice - amountToPlatform - amountToRoyalties, owner);
}
```

Рис. 3.6 Логіка для ставок

Метод виконує розподіл коштів між платформою, правовласником і власником активу. Він забезпечує автоматичну виплату всіх зацікавлених сторін на основі передбачених правил контракту.

Також у програмі наявні класи Emulator і EmulatorWindow, що дозволяють тестувати контракт у середовищі, яке імітує блокчейн. Це забезпечує можливість

перевірки логіки контракту перед його розгортанням у мережі, зменшуючи ризик помилок та збільшуючи надійність розробки.

У програмі також наявні алгоритми шифрування даних які мають досить таки цікаву структру, для кращого розуміння, розглянемо архітектуру програмного коду на рис. 3.7.

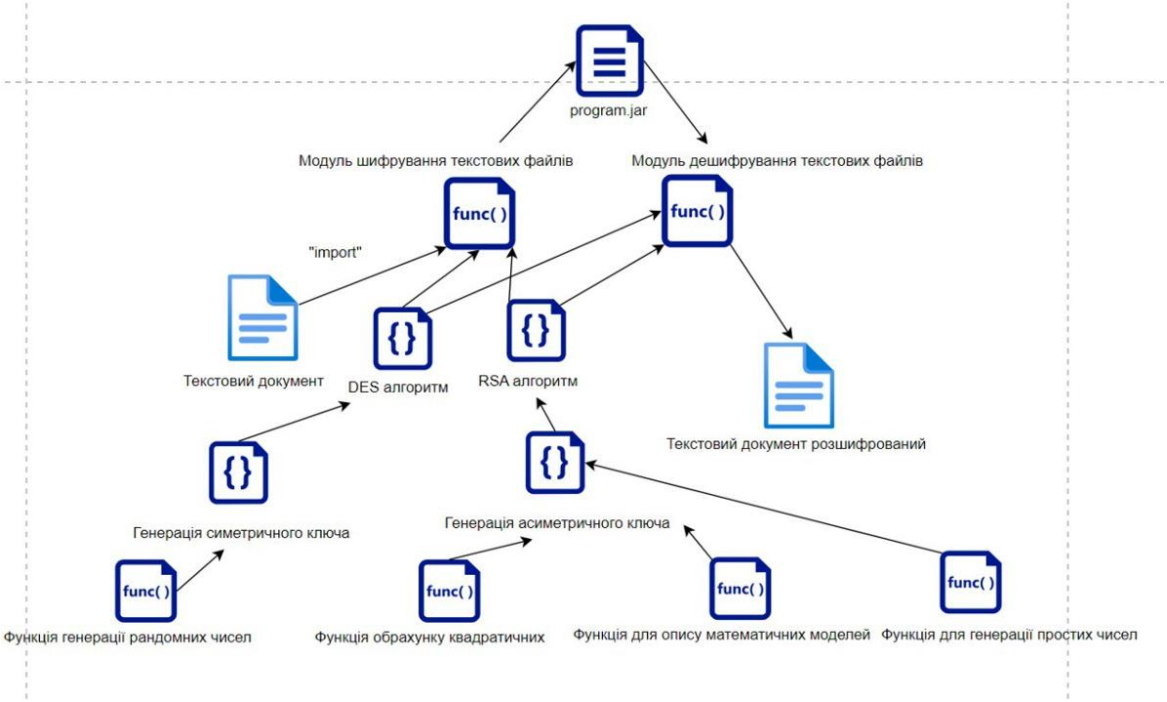

Рис.3.8 Структура коду шифрування інформації

Узагальнена схема алгоритму DES використовується для перетворення блоків даних розміром 64 біти і включає в себе ряд операцій заміни, перестановки та змішування, які застосовуються кілька разів до вхідних даних за допомогою ключа. Шифр DES був досить популярним у свій час, проте згодом було виявлено його обмеження в безпеці через обмежену довжину ключа і зміну технологічного середовища, що призвело до появи більш безпечних алгоритмів шифрування. Детальніше можемо розглянути на рис. 3.9.

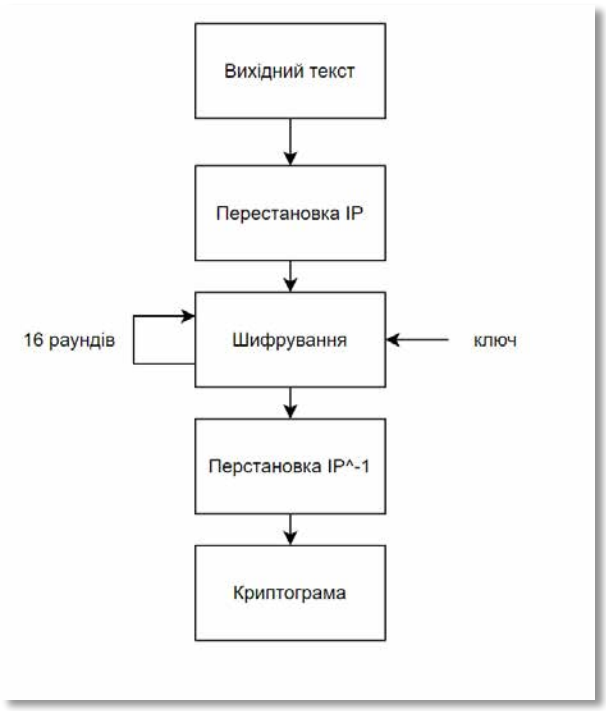

Рис.3.9 - Узагальнена схема шифру DES

Шифрування за алгоритмом DES включає в себе ряд перестановок та операцій, які виконуються над блоками даних. Ключові етапи цього процесу описані у стандарті, і таблиці перестановок, знаходяться в офіційному описі стандарту або документації [9].

Алгоритм DES складається з 16 раундів шифрування, які реалізуються за схемою Фейстеля. Кожен раунд включає в себе ряд операцій, включаючи розширення тексту, застосування раундової функції F та інші маніпуляції. Детальніше можемо побачити на рис. 3.10.

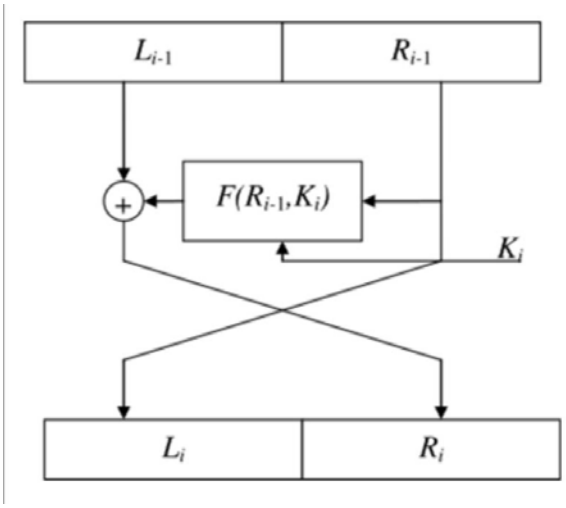

Рис.3.10 - Пряме перетворення за схемою Фейстеля

На етапі розширення тексту правий полублок розширюється до 48 біт за допомогою функції розширення E. Ця функція включає в себе дублювання та перестановку деяких елементів блоку, що приводить до збільшення його розміру.

Після розширення тексту 48-бітний блок стає результатом і залучається до подальших обчислень у межах одного раунду шифрування DES. Функцію розширення блоку можемо побачити на рис. 3.11.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Рис.3.11 - Функція E- розширення блоку

Після розширення отриманий 48-розрядний блок піддається операції побітового добутку (XOR) з раундовим ключем, який буде згенеровано відповідно до схеми, що буде розглянута пізніше.

Далі проводиться операція підстановки за таблицями S_i, які призводять до того, що кожне 6-розрядне вхідне значення замінюється на відповідне 4-розрядне вихідне значення. У результаті цієї операції з вхідних 48 біт отримуємо 32 біти на виході. Таблиці підстановки (S_i) також чітко визначені стандартом і складаються з 16 стовпців і 4 рядків, містять 4-бітові елементи.

Отже, в ході цих операцій відбувається скорочення розміру блоку з 48 до 32 бітів за допомогою побітових операцій XOR та операції підстановки за таблицями Si, що визначені стандартом DES.

3.3 Алгоритми обробки транзакцій та захисту даних.

Основна обробка транзакцій у програмі відбувається за допомогою методів, що відповідають за прийом, обробку та управління транзакціями. Наприклад, у класах Bicho, Cryptoball та SignumArt використовується метод txReceived(), який обробляє транзакцію при її отриманні. Для прикладу розглянемо програмний код у класі SignumArt на рис. 3.12.

```
public void txReceived() {
    if (status == STATUS_FOR_SALE) {
       if (duchStartHeight > ZERO) {
          // Duch auction style, calculate the current price
          currentPrice = startPrice - (getBlockHeight() - duchStartHeight) *
priceDropPerBlock;
          if(currentPrice < reservePrice) {
             currentPrice = reservePrice;
          }
       }
       if (getCurrentTxAmount() >= currentPrice) {
          // Conditions match, let's execute the sale
          pay(); // pay the current owner
          owner = getCurrentTxSender(); // new owner
          status = STATUS_NOT_FOR_SALE;
          sendMessage(owner.getId(), getCurrentTxAmount(), trackNewOwner);

          cancelOfferIfPresent();
          return;
       }
    }
```

Рис.3.12 Реалізація txReceived()

Кожен смарт-контракт має свої особливості обробки транзакцій. Наприклад:

У класі SignumArt метод txReceived() відповідає за управління аукціоном або продажем NFT. Якщо на NFT виставлено ціну, то при отриманні достатньої суми система передає права власності на токен новому власнику, фіксуючи транзакцію та перевіряючи відповідність отриманої суми.

Відповідно у класі Bicho транзакції обробляються для розміщення ставок. В кінці кожного раунду обчислюється переможний номер на основі хешу попереднього блоку, після чого обробляються всі транзакції з попереднього раунду.

Захист даних у системі забезпечується за рахунок використання криптографічних алгоритмів та хешування, що унеможливлює зміну транзакцій

або підробку даних. Для цього в коді використовуються такі методи, як performSHA256 і performSHA256_64. Наприклад, метод performSHA256_64 у класі Contract обчислює хеш для двох вхідних значень типу long, що дозволяє гарантувати унікальність та цілісність даних у блоках. Розглянемо детальніше на рис. 3.13.

```
protected long performSHA256_64(long input1, long input2) {
    Register input = new Register();
    input.value[0] = input1;
    input.value[1] = input2;

    Register ret = performSHA256_(input);
    return ret.getValue1();
}
```

Рис.3.13 Алгоритм захисту даних

Ця функція використовується в таких процесах, як обчислення результатів лотереї або аукціонів, забезпечуючи рандомізацію та захист від маніпуляцій. Наша програма також використовує хеші попередніх блоків для генерації випадкових значень, що робить його менш вразливим до маніпуляцій. Наприклад, у класі Bicho виграшні номери обчислюються на основі хешу попередніх блоків, що забезпечує захист від прогнозованих результатів. Розглянемо детальніше на рис. 3.14.

```
private void roundRun() {
    hash1 = getPrevBlockHash1();
    hash = hash1;
    hashCounter = ONE;
    while (hashCounter < HASH_BLOCKS) {
       sleep(ONE);
       hash *= TWO; // зрушення на один біт
       hash += (getPrevBlockHash1() & ONE);
       hashCounter += ONE;
    }
    hash = performSHA256_64(hash, hash1);
}
```

Рис.3.14 Генерація випадкових чисел

Цей код генерує випадкове значення, використовуючи хеш попереднього блоку, що ускладнює передбачення майбутніх результатів. Це забезпечує справедливість у випадкових процесах, таких як аукціони або лотереї, захищаючи систему від шахрайства. Кожна транзакція, що надходить до контракту, перевіряється на відповідність вимогам. Наприклад, для аукціону перевіряється, чи сума транзакції відповідає мінімально необхідній ставці.

3.4 Опис елементів інтерфейсу системи

У нашому проекті реалізовано 5 класів для реалізації UI програмної системи, що можемо побачити на рис. 3.15.

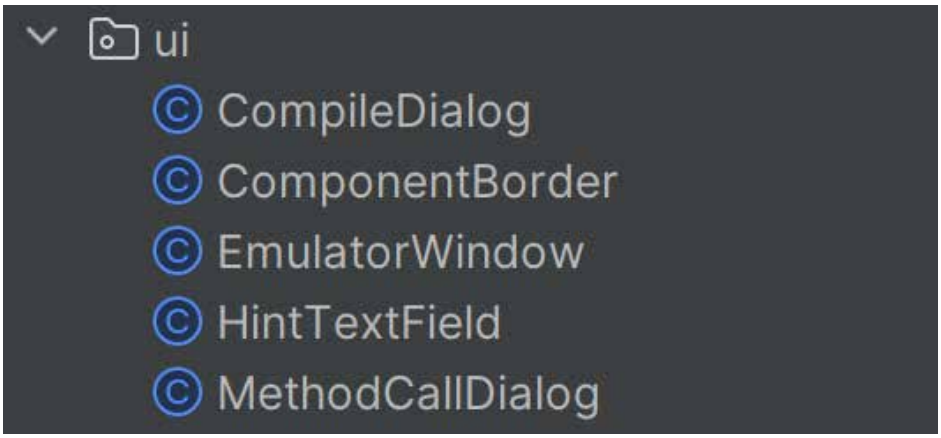

Рис.3.15 Класи системи які відповідають за інтерфейс користувача

Розглянемо опис кожного класу детальніше у таблиці 3.1.

Таблиця 3.1

Опис класів які відповідають за інтерфейс програмної системи

Клас	Опис
CompileDialog	Реалізує діалогове вікно для компіляції, яке може включати функціонал вибору файлів, перевірки синтаксису та запуску процесу компіляції з відображенням помилок.
ComponentBorder	Відповідає за відображення меж компонентів інтерфейсу, налаштовує зовнішній вигляд меж (колір, товщина, стиль).
EmulatorWindow	Основне вікно емулятора, яке надає доступ до інструментів тестування і налагодження смарт-контрактів, візуалізації транзакцій, запуску тестів і відображення результатів.
HintTextField	Реалізовує текстове поле з підказкою, що зникає при введенні даних, допомагає користувачам зрозуміти призначення полів.
MethodCallDialog	Створює діалогове вікно для виклику методів або функцій, дозволяє вибирати методи, вводити параметри та переглядати результати виконання.

Перейдемо до огляду самого інтерфейсу. Розглянемо рисунок 3.16.

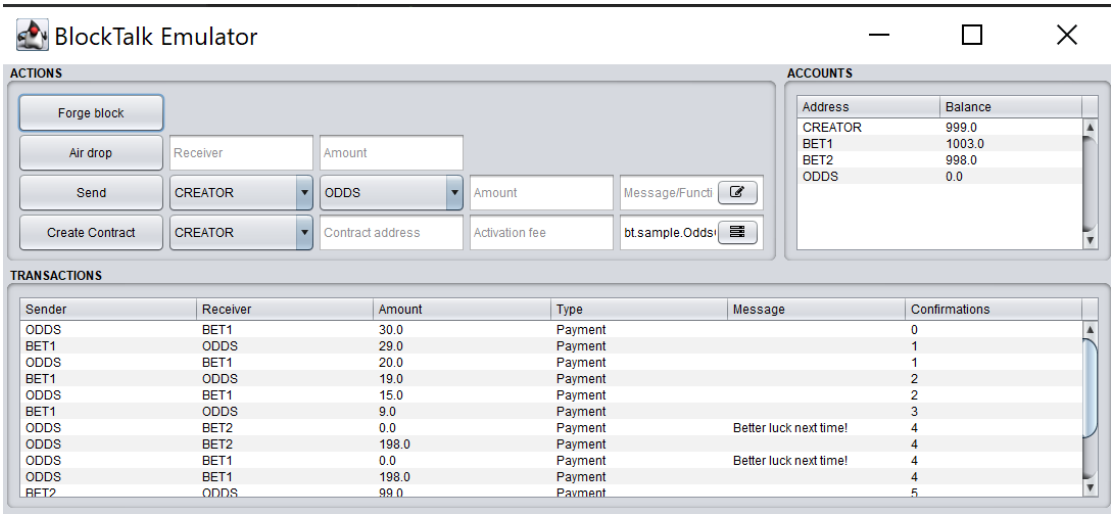

Рис.3.16 Інтерфейс системи

Інтерфейс системи "BlockTalk Emulator" складається з кількох основних блоків: "Actions" (Дії), "Accounts" (Акаунти) та "Transactions" (Транзакції). У блоці "Actions", розташованому у верхній частині інтерфейсу, містяться функції для управління транзакціями та операціями в системі. Основні дії включають можливість створення блоку (Forge block), надсилання коштів через функцію Air drop, відправлення коштів між акаунтами за допомогою функції Send, а також створення контрактів через функцію Create Contract, де можна вказати адресу відправника, адресу контракту та комісію за активацію.

Блок "Accounts" розташований праворуч і містить таблицю з інформацією про акаунти, що доступні в системі. У таблиці відображені адреси акаунтів і їхні баланси, що дозволяє швидко побачити стан кожного акаунта та доступні залишки коштів.

Блок "Transactions", розташований у нижній частині інтерфейсу, показує історію транзакцій. Таблиця містить колонки з інформацією про відправника транзакції, отримувача, суму переданих коштів, тип транзакції (наприклад, "Payment"), текст повідомлення, яке супроводжує транзакцію (наприклад, "Better luck next time!"), а також кількість підтверджень для кожної транзакції.

Цей інтерфейс забезпечує доступ до основних функцій блокчейн-емуляції, дозволяючи користувачам керувати акаунтами, проводити транзакції, створювати блоки та працювати з контрактами у зручному та наглядному форматі. Розглянемо на рис. 3.17 наступний елемент системи.

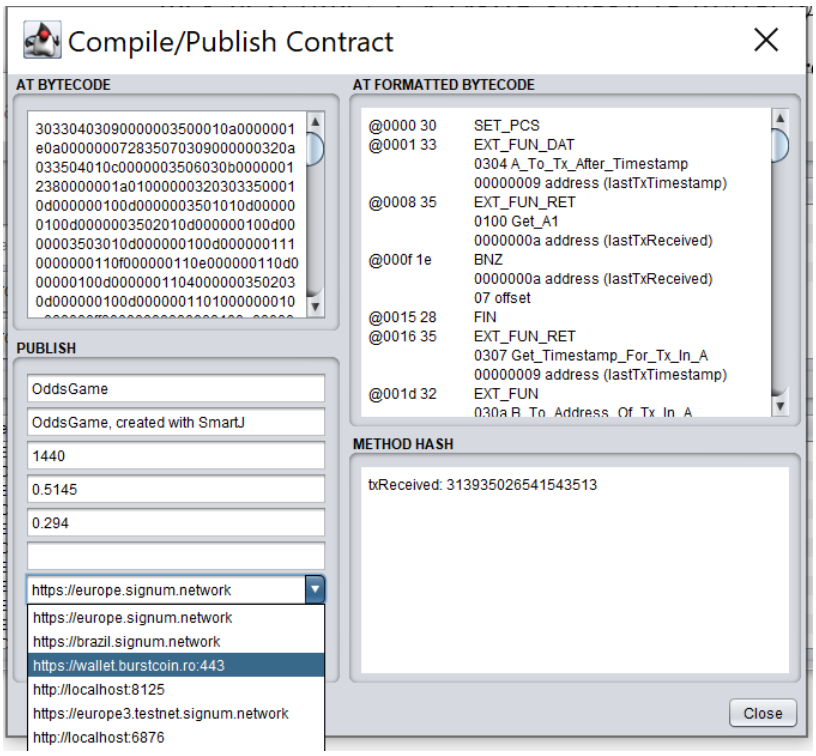

Рис.3.17 Компіляція і публікація смарт-контракту

Ліва верхня частина містить поле "AT Bytecode", в якому представлений скомпільований байткод смарт-контракту у вигляді послідовності чисел і символів. Це необроблене представлення коду, готове до завантаження в блокчейн. Справа від цього поля знаходиться панель "AT Formatted Bytecode", де відображено байткод у форматі інструкцій. Цей формат спрощує читання та перевірку коду, показуючи кожну операцію, яка виконується в рамках контракту, з відповідними командами і адресами. У нижній лівій частині блоку знаходиться розділ "Publish". Він дозволяє користувачеві ввести ім'я контракту ("OddsGame") та опис ("OddsGame, created with SmartJ"). Також можна встановити параметри, такі як час існування контракту (наприклад, "1440"), комісію за виконання та інші параметри, що стосуються фінансових аспектів контракту. Нижче панелі "Publish" розташоване випадаюче меню, яке надає список URL-адрес для вибору блокчейн-вузла, куди буде завантажений контракт. Це дозволяє користувачеві обрати тестову або основну мережу, залежно від потреб розгортання.

Праворуч від панелі "Publish" розташоване поле "Method Hash". У ньому відображено хеш функції контракту (в даному випадку, "txReceived"), що спрощує ідентифікацію конкретних методів під час виклику функцій контракту.

4 ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ СИСТЕМИ

4.1 Тестування функціональності системи.

Для тестування системи ми створили багато тестових випадків, що охоплюють різні аспекти її роботи. Кожен із тестових класів, представлений у списку, відповідає за перевірку окремих функціональних або технічних можливостей системи. Наприклад, такі класи, як OddsGame, Crowdfund, Auction, PaymentChannel, NFT2, та інші забезпечують моделювання різних сценаріїв використання для гарантії стабільної роботи смарт-контрактів у різноманітних умовах. Кожен тестовий випадок спрямований на виявлення потенційних помилок, оптимізацію процесів, а також перевірку безпеки і продуктивності системи в цілому. Класи можемо побачити на рис. 4.1.

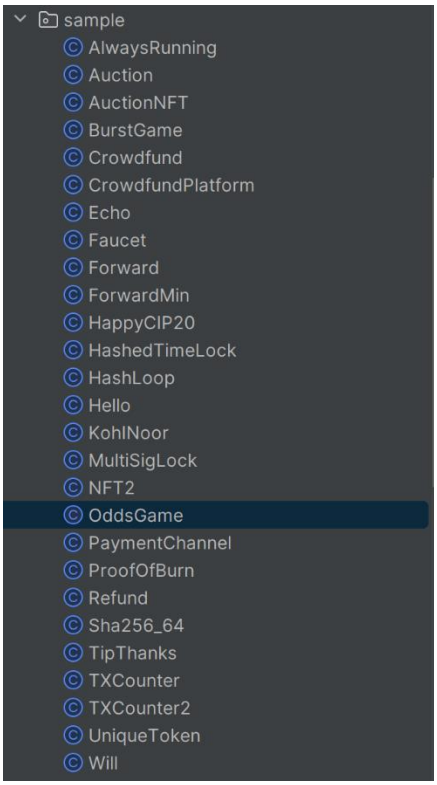

Рис.4.1 Класи з тестовими випадками

Для прикладу розглянемо кілька класів для тестування. Почнемо з прикладу BurstGame. Програмний код тестового випадку розглянемо на рис.4.2.

```
@TargetCompilerVersion(CompilerVersion.v0_0_0)
public class BurstGame extends Contract {

    Address challenger;

    long challengerAmount;
    long creatorAmount;
    long balance;
    long blockHash;

    static final String DEV_ADDRESS = "BURST-JJQS-MMA4-GHB4-4ZNZU";
    public void txReceived(){
       challenger = getCurrentTx().getSenderAddress();
       if(challenger == getCreator()){
          if(getCurrentTx().getAmount() == 0){
             // If creator sends a message with 0 amount (exactly the activation fee)
             // we withdraw the current balance
             sendBalance(challenger);
          }
          // do nothing, creator is just increasing his amount
          return;
       }
       challengerAmount = getCurrentTx().getAmount();
       balance = getCurrentBalance();
       creatorAmount = balance - challengerAmount;
       // sleep two blocks
       sleep(2);
       blockHash = getPrevBlockHash().getValue1();
       blockHash &= 0x0FFFFFFFFFFFFFFFL; // avoid negative values
       blockHash %= balance;

       if(blockHash < creatorAmount){
          // tip developer with 0.5 percent
          sendAmount(balance/200, parseAddress(DEV_ADDRESS));
          // creator wins
          sendBalance(getCreator());
       }
       else {
          // tip developer with 1 percent
          sendAmount(balance/100, parseAddress(DEV_ADDRESS));
          // challenger wins
          sendBalance(challenger);
       }
    }
   public static void main(String[] args) throws Exception {
       // some initialization code to make things easier to debug
       Emulator emu = Emulator.getInstance();
       Address creator = Emulator.getInstance().getAddress("CREATOR");
       Address challenger = Emulator.getInstance().getAddress("CHALLENGER");
       emu.airDrop(creator, 1000 * ONE_BURST);
       emu.airDrop(challenger, 1000 * ONE_BURST);
       Address odds = Emulator.getInstance().getAddress("GAME");
       emu.createConctract(creator, odds, BurstGame.class, ONE_BURST);
       emu.forgeBlock();
       emu.send(challenger, odds, 100*ONE_BURST);
       emu.send(challenger, odds, 100*ONE_BURST);
       emu.send(challenger, odds, 100*ONE_BURST);
       emu.send(challenger, odds, 100*ONE_BURST);
       emu.send(challenger, odds, 100*ONE_BURST);
       emu.send(challenger, odds, 100*ONE_BURST);
       emu.send(challenger, odds, 100*ONE_BURST);
       emu.send(challenger, odds, 100*ONE_BURST);
       emu.send(challenger, odds, 100*ONE_BURST);
       emu.send(challenger, odds, 100*ONE_BURST);
       emu.forgeBlock();
       emu.forgeBlock();
       emu.send(challenger, odds, 10*ONE_BURST);
       emu.forgeBlock();
       emu.send(challenger, odds, 20*ONE_BURST);
       emu.forgeBlock();
       emu.forgeBlock();
       emu.send(challenger, odds, 30*ONE_BURST);
       emu.forgeBlock();
       emu.forgeBlock();
       new EmulatorWindow(BurstGame.class);
    }
}
```

Рис.4.2 Програмний код тестового класу BurstGame

Нас зустрічає програма з даними, що можемо побачити на рис. 4.3.

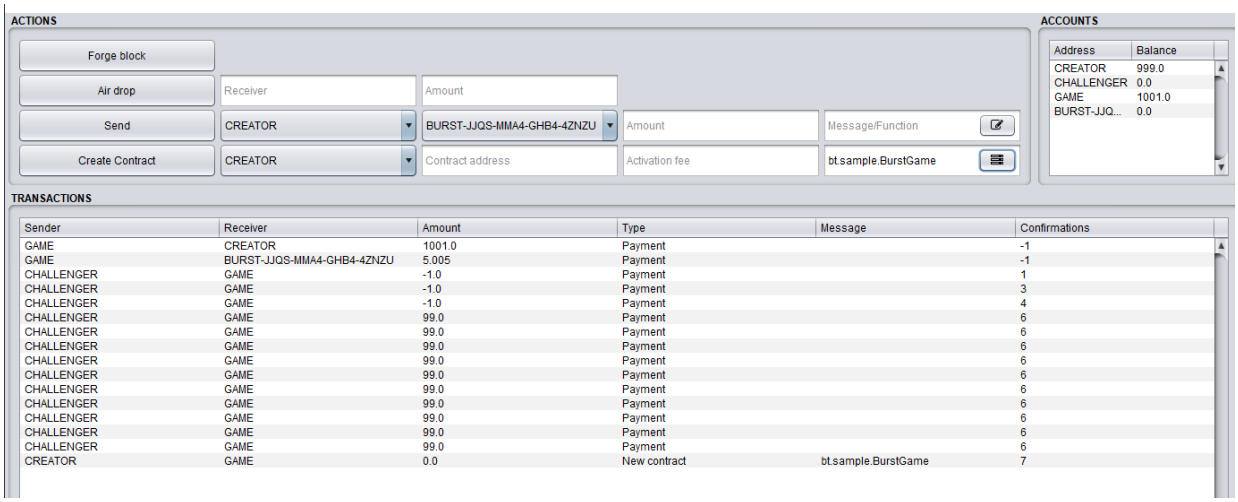

Рис.4.3 Програмна реалізація BurstGame

У верхній частині інтерфейсу є панель дій, що дозволяє виконувати ключові операції, такі як «Forge block», «Air drop», «Send», і «Create Contract». З правого боку відображено список облікових записів із балансами, включаючи облікові записи CREATOR, CHALLENGER, GAME, і певний адрес BURST-JJQS-MMA4-GHB4-4ZNZU.

У таблиці транзакцій наведено історію операцій між обліковими записами, де CHALLENGER надсилає суми до GAME. Типи транзакцій переважно позначені як «Payment», що свідчить про передачу коштів між учасниками. Стовпець «Confirmations» відображає статус підтверджень для кожної транзакції.

З цього зображення видно, що відбувалося тестування функціональності контракту BurstGame, а саме перевірка сценарію взаємодії між CHALLENGER і GAME. Ці транзакції дозволяють перевірити, як смарт-контракт обробляє ставки від учасників, чи коректно змінюються баланси і чи обробляються підтвердження для кожної операції. Розглянемо контракт на рис. 4.4.

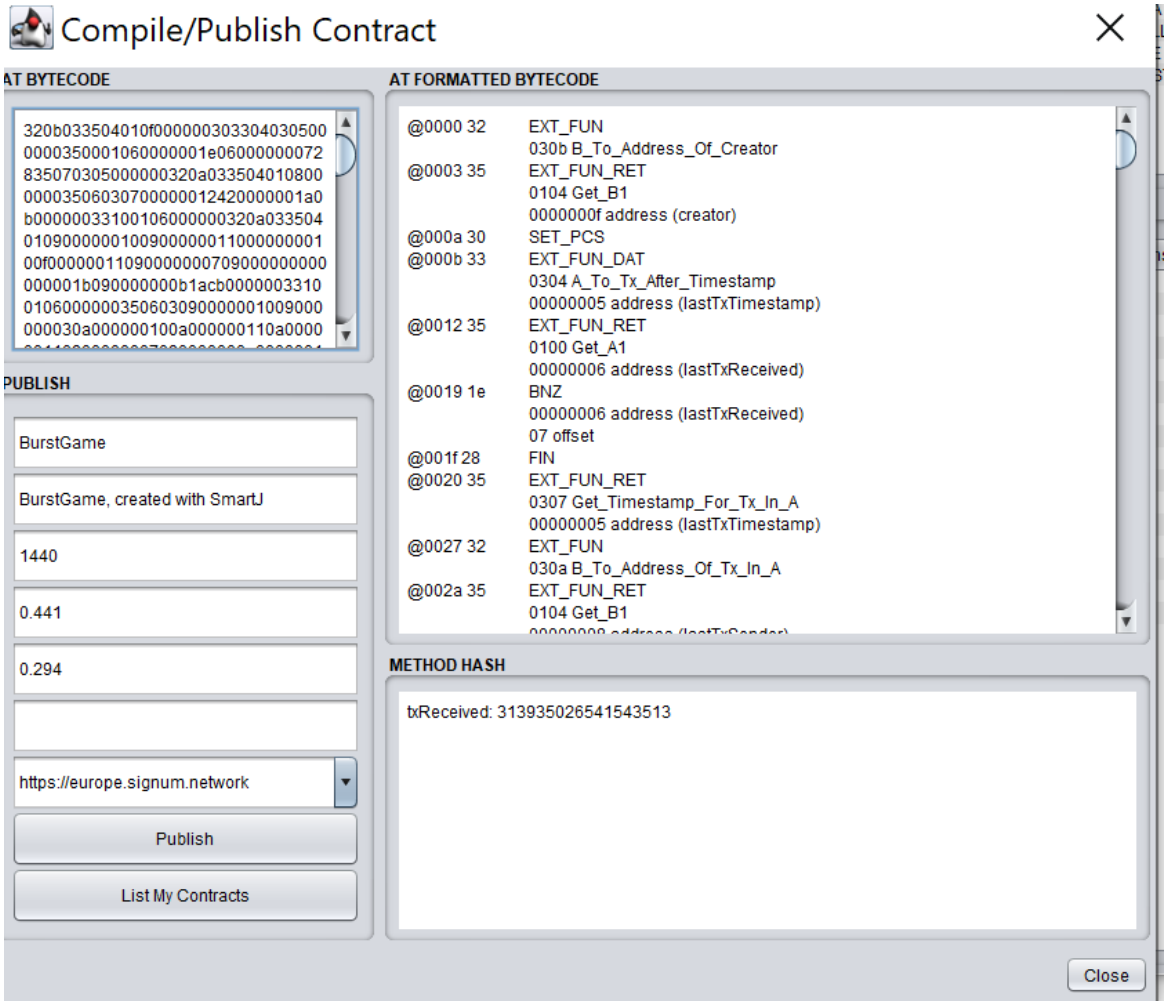

Рис.4.4 Інтерфейс вкладки Publish Contract

Ліва частина вікна включає блок AT BYTECODE, де відображається байт-код контракту. Цей код є скомпільованим представленням логіки смарт-контракту і використовується для його виконання на блокчейні.

Посередині знаходиться секція AT FORMATTED BYTECODE, де байт-код представлений у форматі асемблерних інструкцій. Тут можна побачити окремі команди, такі як EXT_FUN, SET_PCS, і FIN, які відповідають за виконання різних функцій у контракті. Ці інструкції деталізують функціональність контракту і вказують на роботу з різними змінними та адресами у блокчейні.

У розділі PUBLISH користувач може вказати назву контракту, короткий опис, а також значення параметрів, таких як час підтвердження (1440), комісії за публікацію і транзакції (0.441 та 0.294). Також тут є випадаючий список для вибору мережі, в яку буде опубліковано контракт, у цьому випадку — Signum Network.

Справа внизу, в секції METHOD HASH, міститься хеш-значення для методу txReceived, яке є унікальним ідентифікатором для цього методу контракту. Це значення використовується для виклику конкретного методу в контракті, щоб забезпечити його коректну ідентифікацію під час виконання.

Розглянемо найскладніший в плані реалізації тестовий випадок NFT2, програмний код якого представлений у ДОДАТОК Г. При запуску програми можемо побачити інтерфейс, представлений на рис. 4.5.

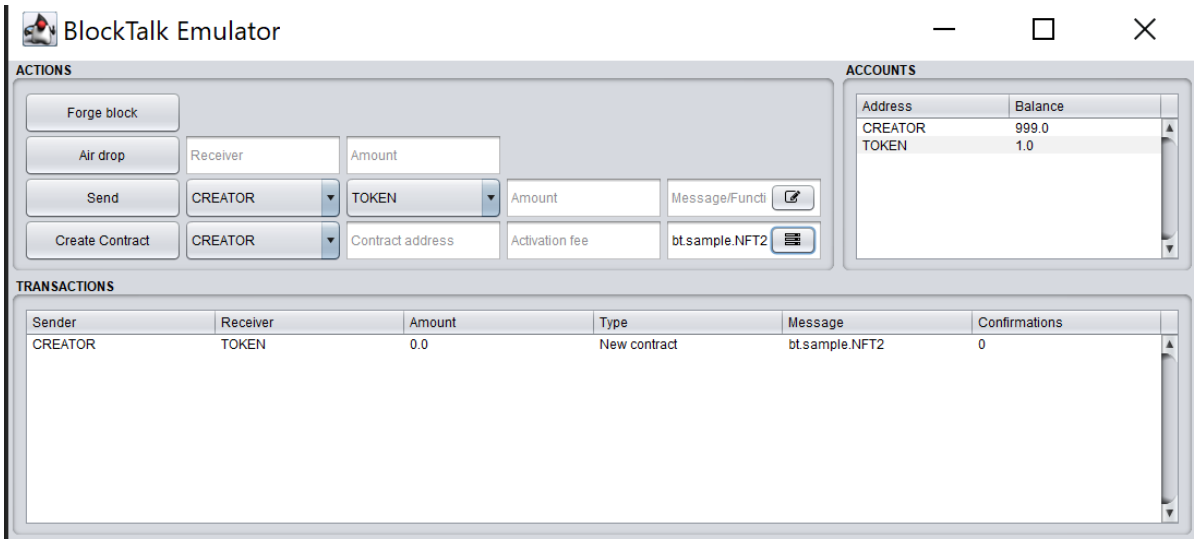

Рис. 4.5 Інтерфейс програмної системи тестового випадку NFT2

У цьому прикладі набагато цікавішим є самий контракт який можемо побачити на рис. 4.6.

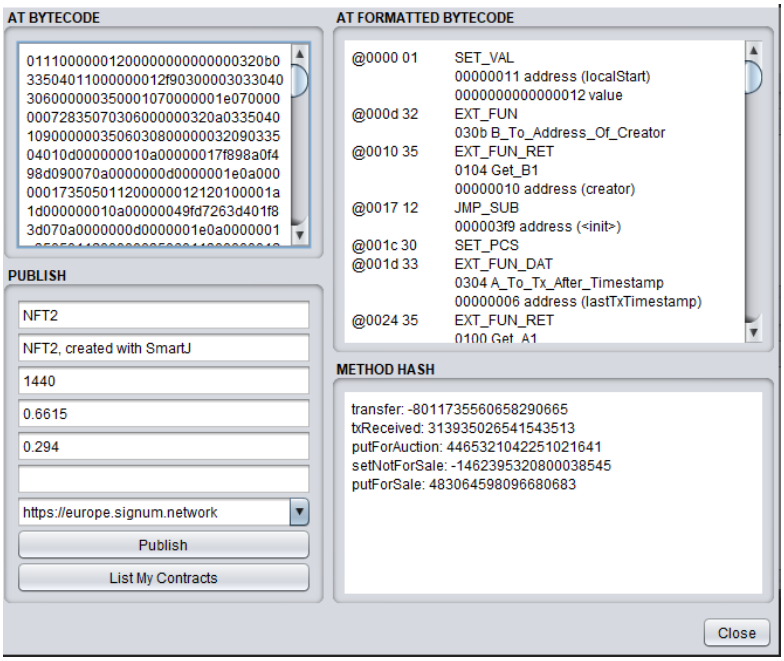

Рис.4.6 Інтерфейс і публікація контракту NFT2

Праворуч, у розділі AT FORMATTED BYTECODE, байт-код форматовано у вигляді асемблерних команд для покращеного розуміння кроків виконання контракту. Інструкції, такі як SET_VAL, EXT_FUN, та JMP_SUB, відображають логіку контракту та вказують на взаємодію з різними змінними й адресами. Це дозволяє детально переглянути процеси, які відбуваються під час виконання контракту.

У розділі PUBLISH користувач може задати назву контракту, короткий опис, кількість блоків для підтвердження (1440), а також комісії за публікацію та виконання контракту (0.6615 і 0.294 відповідно). Також можна вибрати блокчейн-мережу для публікації контракту, в даному випадку — Signum Network.

Нижче, у розділі METHOD HASH, відображаються хеші для різних методів контракту, таких як transfer, txReceived, putForAuction, setNotForSale, і putForSale. Ці хеш-значення використовуються для ідентифікації методів контракту при викликах, що дозволяє точно звертатися до відповідних функцій під час тестування або виконання контракту на блокчейні.

Інтерфейс дозволяє натиснути кнопку Publish для розгортання контракту у вибраній мережі або переглянути інші контракти, опубліковані користувачем, через опцію List My Contracts.

4.2 Аналіз тестування і оптимізація

Аналіз тестування та оптимізація — це невід'ємні етапи розробки програмного забезпечення, які впливають на його надійність, продуктивність та зручність у використанні. Після написання коду система проходить через ретельне тестування, що дозволяє виявити потенційні помилки, слабкі місця й можливості для вдосконалення. Однак тестування — це не лише пошук помилок, але й глибоке дослідження функціональності для забезпечення відповідності системи вимогам користувача та очікуваній поведінці в реальних умовах.

Саме на цьому етапі аналізується поведінка системи під різними навантаженнями, з різними сценаріями використання та в умовах обмежених ресурсів. Оптимізація ж допомагає досягти максимальної ефективності роботи, зменшити затримки й забезпечити швидку обробку даних. Водночас, цей процес вимагає підходу, де зміни вносяться обережно, аби не порушити існуючу функціональність. Розглянемо тестові випадки, відповідно для початку до розгляду таблиця 4.1.

Таблиця 4.1

Тестування функціональності створення контракту

Тестовий випадок	Опис	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
Створення нового контракту	Тестування створення контракту	Тип контракту: NFT2, адреса платформи, комісія	Контракт успішно створено	+	Pass/Fail
Створення контракту з помилковими даними	Перевірка, якщо вхідні дані некоректні	Тип контракту: NFT2, комісія: негативна	Повертається помилка про некоректні дані	+	Pass/Fail
Перевірка ідентифікації контракту	Перевірка унікальності контракту при створенні	Ідентифікатор: унікальний	Контракт отримує унікальний ID	+	Pass/Fail

Перейдемо до наступного тестового випадку представленого у таблиці 4.2.

Таблиця 4.2

Тестування функціональності транкзакцій

Тестовий випадок	Опис	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
Відправка транзакції	Тестування відправки транзакції між рахунками	Відправник: "CREATOR", отримувач: "GAME", сума: 100	Транзакція успішно надіслана та оброблена	+	Pass/Fail
Неправильний формат даних	Відправка транзакції з некоректними даними	Отримувач: "НевірнийID", сума: -50	Помилка в обробці транзакції	+	Pass/Fail
Перевірка балансу	Перевірка оновлення балансу після транзакції	Відправник: "CREATOR", отримувач: "GAME", сума: 200	Баланси обох рахунків оновлено коректно	+	Pass/Fail

Наступною буде перевірка сценаріїв аукціонів представлена у таблиці 4.3.

Таблиця 4.3

Тестування сценаріїв аукціонів

Тестовий випадок	Опис	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
Старт аукціону	Створення аукціону з мінімальною ставкою	Початкова ставка: 50, тайм-аут: 1 год	Аукціон успішно створено	+	Pass/Fail
Збільшення ставки	Тестування підвищення ставки	Попередня ставка: 50, нова ставка: 60	Ставку успішно підвищено	+	Pass/Fail
Завершення аукціону	Перевірка завершення аукціону після тайм-ауту	Тайм-аут: пройдено	Власник змінюється, фінали завершені	+	Pass/Fail

І останнім сценарійом, буде тестування функціональності NFT-токенів, що представлено у таблиці 4.4.

Таблиця 4.4

Тестування функціональності NFT-токенів

Тестовий випадок	Опис	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
Створення NFT	Перевірка створення NFT-токену	Назва: "ArtPiece", ціна: 100 SIGNA	NFT успішно створено	+	Pass/Fail
Трансфер NFT	Передача NFT іншому користувачу	Власник: "USER1", новий власник: "USER2"	NFT успішно передано новому власнику	+	Pass/Fail
Встановлення продажу	Встановлення ціни на продаж NFT	NFT ID: 1, нова ціна: 200 SIGNA	Ціна на NFT встановлена	+	Pass/Fail

Оптимізація системи спрямована на підвищення її продуктивності, зниження витрат ресурсів та забезпечення плавної роботи при масштабуванні. Першим кроком є аналіз існуючих компонентів і виявлення "вузьких місць" — елементів, які можуть уповільнювати роботу системи, особливо при високих навантаженнях. Це включає перевірку часу обробки транзакцій, ефективності криптографічних операцій, а також використання пам’яті і процесорних ресурсів.

Одним із ключових аспектів є оптимізація алгоритмів обробки транзакцій. У випадку з PoC (Proof of Capacity), реалізованому на платформі Signum, важливо мінімізувати енергоспоживання та витрати на майнінг. Це досягається за рахунок зменшення складності обчислень і більш раціонального використання дискового простору для зберігання хешів. Завдяки PoC, система знижує потребу в постійно активному високопродуктивному обладнанні, що позитивно впливає на екологічність і загальні витрати.

Щодо контрактів, оптимізація полягає у зменшенні розміру байт-коду смарт-контрактів, що дозволяє зекономити ресурси при виконанні операцій на

блокчейні. Важливим аспектом є мінімізація кількості методів і параметрів, що передаються у викликах контрактів, адже кожен метод споживає певний обсяг ресурсів на його обробку та зберігання.

Додатково, для оптимізації інтерфейсу користувача використовуються асинхронні обробки запитів, що дозволяє уникнути блокувань при одночасній роботі декількох користувачів. Це забезпечує швидкий відгук системи та комфортне користування без затримок.

Ще одним важливим аспектом оптимізації є використання кешування даних, особливо для часто запитуваних операцій, таких як баланс рахунку або історія транзакцій. Кешування дозволяє зменшити навантаження на основну базу даних і прискорити доступ до інформації, що важливо при високих навантаженнях.

4.3 Оцінка ефективності системи

Оцінка ефективності системи базується на кількох ключових параметрах, що відображають її продуктивність, стійкість до навантажень та зручність для користувачів. По-перше, завдяки використанню алгоритму PoC (Proof of Capacity) платформа Signum демонструє високу енергоефективність. На відміну від алгоритмів PoW (Proof of Work), які вимагають значних обчислювальних потужностей і споживають багато електроенергії, PoC дозволяє суттєво знизити витрати енергії та підтримує майнінг навіть на базовому обладнанні. Це забезпечує екологічну стійкість системи, знижує витрати на обслуговування і робить майнінг доступним для ширшого кола користувачів.

З точки зору швидкості обробки транзакцій, система здатна ефективно масштабуватися завдяки асинхронним методам обробки даних, що мінімізує затримки при взаємодії користувачів із смарт-контрактами. Використання кешування і оптимізації алгоритмів зберігання та обробки даних забезпечує високу швидкодію, особливо під час виконання частих запитів, таких як перевірка балансу чи перевірка статусу транзакції. Це підвищує загальну продуктивність та швидкість відповіді системи, що є важливим для забезпечення безперервної та плавної роботи платформи навіть при високих навантаженнях.

Крім того, система демонструє високий рівень безпеки завдяки використанню надійних криптографічних алгоритмів. Бібліотека BouncyCastle, інтегрована для забезпечення криптографічного захисту, гарантує надійність підпису та верифікації транзакцій, а також захист даних від несанкціонованого доступу. Це є критично важливим для збереження конфіденційності та цілісності інформації у блокчейні.

Загалом, оцінка ефективності системи показує, що вона досягає оптимального балансу між продуктивністю, енергоефективністю, безпекою та зручністю використання. Завдяки цим властивостям система може функціонувати стабільно і безпечно при високих навантаженнях, що робить її ефективним і надійним рішенням для широкого спектра користувачів.

ВИСНОВКИ

У цій дипломній роботі було проведено комплексний аналіз сучасних систем управління смарт-контрактами, виявлено їхні слабкі місця та обмеження. Це дало можливість чітко визначити потребу в новій, більш надійній системі, що здатна враховувати особливості децентралізованого середовища та вимоги щодо безпеки й ефективності. На основі проведеного дослідження було сформульовано вимоги до розробки системи управління смарт-контрактами з використанням блокчейн-технологій, що забезпечує децентралізоване зберігання даних, прозорість транзакцій та мінімізацію ризиків шахрайства.

У процесі роботи розроблено архітектуру системи, що враховує специфіку децентралізації та потреби користувачів у захисті даних. Архітектура побудована з урахуванням необхідності зберігати дані в блокчейні та інтеграції різноманітних криптографічних інструментів для захисту транзакцій.

Реалізований прототип системи було успішно протестовано на відповідність встановленим вимогам. Тестування продемонструвало ефективність системи у виконанні транзакцій, захисті даних, а також стійкість до потенційних загроз, пов'язаних із децентралізованим середовищем. Аналіз результатів тестування підтвердив відповідність реалізованої системи поставленим завданням, що свідчить про її готовність до подальшого розгортання.

Запропоновано кроки для інтеграції системи в різноманітні сфери, де використання смарт-контрактів може підвищити ефективність процесів, а також запропоновано напрямки для покращення продуктивності та розширення функціональних можливостей. Таким чином, розроблена система управління смарт-контрактами на основі блокчейн-технологій є перспективним інструментом для вирішення завдань у сфері децентралізованих фінансів, управління активами та цифрової комерції.

СПИСКИ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бутерин В. Ефіріум: наступне покоління криптовалюти та децентралізованого додатка. 2014. URL: https://ethereum.org/whitepaper.

2. Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. URL: https://bitcoin.org/bitcoin.pdf.

3. Szabo N. Smart Contracts: Building Blocks for Digital Markets. 1996. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html.

4. Стегній А. В. Дослідження сучасних підходів до побудови блокчейн-систем // Інформаційні технології та комп'ютерна інженерія. 2021. № 2. С. 45–53.

5. Antonopoulos A. Mastering Bitcoin: Unlocking Digital Cryptocurrencies. O'Reilly Media, 2017. 298 p.

6. Мартинов І. І. Блокчейн та смарт-контракти: нові можливості для бізнесу // Бізнес Інформ. 2020. № 9. С. 66–71.

7. Tikhomirov S. Security Analysis of Ethereum Smart Contracts. 2017. URL: https://arxiv.org/abs/1703.03994.

8. Харченко В. С. Розробка та впровадження блокчейн технологій в Україні // Науковий вісник ХНТУ. 2021. № 4. С. 90–98.

9. Mougayar W. The Business Blockchain: Promise, Practice, and the Application of the Next Internet Technology. Wiley, 2016. 208 p.

10. Кравець В. В., Мартинов А. О. Смарт-контракти: концепція, технологія та перспективи застосування // Вісник ХНУ імені В.Н. Каразіна. 2019. № 10. С. 121–128.

11. Popov S. The Tangle. 2018. URL: https://iota.org/IOTA_Whitepaper.pdf.

12. Кузнєцов О. М. Енергоефективність та безпека блокчейн-алгоритмів // Вісник Національного технічного університету «ХПІ». 2022. № 1. С. 45–51.

13. Greenspan G. Multichain Private Blockchain: White Paper. 2015. URL: https://www.multichain.com/download/MultiChain-White-Paper.pdf.

14. Шахов Д. В., Павленко М. П. Блокчейн та кібербезпека: перспективи застосування в Україні // Інформаційні технології та безпека. 2020. № 5. С. 112–119.

15. Wood G. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Yellow Paper, 2014. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

16. Singh A., Singh K. Blockchain Basics. Springer, 2019. 90 p.

17. Шевченко О. О. Актуальні питання розробки смарт-контрактів у децентралізованих системах // Вісник Київського національного університету ім. Тараса Шевченка. 2021. № 3. С. 70–78.

18. Pilkington M. Blockchain Technology: Principles and Applications. Research Handbook on Digital Transformations. Edward Elgar Publishing, 2016. P. 225–253.

19. Мірошниченко Д. В. Блокчейн-технології в електронному управлінні: проблеми і перспективи розвитку // Державне управління: удосконалення та розвиток. 2021. № 10. С. 38–44.

20. Bashir I. Mastering Blockchain: Unlocking the Power of Cryptocurrencies, Smart Contracts, and Decentralized Applications. Packt Publishing, 2020. 586 p.

21. Петров Д. І., Іванов О. П. Смарт-контракти у фінансових технологіях: застосування та виклики // Фінансовий ринок. 2022. № 2. С. 54–61.

22. Gupta M. Blockchain for Dummies. Wiley, 2017. 212 p.

23. Климчук С. В., Шевченко А. В. Проблеми реалізації смарт-контрактів на базі блокчейн-технологій // Економіка та право. 2020. № 6. С. 99–106.

24. Risius M., Spohrer K. A Blockchain Research Framework. Business & Information Systems Engineering, 2017. Vol. 59. No. 6. P. 385–409.

25. Романюк А. І., Павлова М. В. Вплив блокчейн-технології на розвиток фінансових ринків // Інноваційна економіка. 2021. № 12. С. 103–109.

26. Buterin V., Griffith V. Casper the Friendly Finality Gadget. 2017. URL: https://arxiv.org/abs/1710.09437.

27. Ткаченко В. М., Кириченко Л. О. Можливості використання блокчейн технологій у сфері медицини // Здоров'я нації. 2022. № 1. С. 45–51.

28. Yaga D., Mell P., Roby N., Scarfone K. Blockchain Technology Overview. NIST, 2018. URL: https://nvlpubs.nist.gov/nistpubs/ir/2018/NIST.IR.8202.pdf.

29. Drescher D. Blockchain Basics: A Non-Technical Introduction in 25 Steps. Apress, 2017. 255 p.

30. Байда С. В. Проблеми та перспективи впровадження блокчейн-технологій в Україні // Наукові записки. 2021. № 3. С. 60–67.

ДОДАТОК А

ЛІСТИНІНГ КОДУ КЛАСУ SIGNUMART

```
@TargetCompilerVersion(CompilerVersion.v0_0_0)
public class SignumArt extends Contract {

    Address owner;
    Address platformAddress;
    long platformFee;
    long royaltiesFee;
    Address royaltiesOwner;

    Address tracker;

    long status;
    long currentPrice;
    Timestamp auctionTimeout;
    Address highestBidder;

    long totalTimesSold;
    long totalBidsReceived;
    long totalRoyaltiesFee;
    long totalPlatformFee;

    long amountToRoyalties;
    long amountToPlatform;

    private static final long ZERO = 0;
    private static final long STATUS_NOT_FOR_SALE = ZERO;
    private static final long STATUS_FOR_SALE = 1;
    private static final long STATUS_FOR_AUCTION = 2;
    private static final long NEW_AUCTION_BID = 3;
    private static final long NEW_OWNER = 4;
    private static final long TRANSFER_OWNERSHIP = 5;
    private static final long THOUSAND = 1000;

    /** Use a contract fee of 0.3 SIGNA */
    public static final long CONTRACT_FEES = 30000000;

    /**
     * Transfers the ownership of this token.
     *
     * Only the current owner can transfer the ownership.
     *
     * @param newOwner
     */
    public void transfer(Address newOwner) {
       if (owner.equals(this.getCurrentTxSender())) {
          // only the current owner can transfer
          owner = newOwner;
          sendMessage(TRANSFER_OWNERSHIP, owner.getId(), tracker);
       }
    }

    /**
     * Transfers the royalties ownership of this token.
     *
     * Only the current royalties owner can transfer the ownership.
     *
     * @param newRoyaltiesOwner
     */
    public void transferRoyalties(Address newRoyaltiesOwner) {
       if (royaltiesOwner.equals(this.getCurrentTxSender())) {
          // only the current owner can transfer
          royaltiesOwner = newRoyaltiesOwner;
       }
    }
```

```
    /**
     * Cancels an open for sale or auction and sets it as not for sale.
     */
    public void setNotForSale(){
       if (highestBidder==null && owner.equals(this.getCurrentTxSender())) {
          // only if there is no bidder and it is the current owner
          status = STATUS_NOT_FOR_SALE;
          sendMessage(status, tracker);
       }
    }

    /**
     * Put this token for sale for the given price.
     *
     * Buyer needs to transfer at least the asked amount.
     *
     * @param priceNQT the price in NQT==1E-8 SIGNA (buyer needs to transfer
at least
     *                 this amount + gas fees)
     */
    public void putForSale(long priceNQT) {
       if (highestBidder==null && owner.equals(this.getCurrentTxSender())) {
          // only if there is no bidder and it is the current owner
          status = STATUS_FOR_SALE;
          currentPrice = priceNQT;
          sendMessage(status, owner.getId(), currentPrice, ZERO, tracker);
       }
    }

    /**
     * Put this token for auction with the minimum bid price.
     *
     * Bidders need to transfer more than current highest to become the new
     * highest bidder. Previous highest bidder is refunded in case of a new
     * highest bid arrives.
     *
     * @param priceNQT the minimum bid price in NQT==1E-8 SIGNA (buyer needs
to
     *                 transfer at least this amount + contract fees)
     * @param timeout  how many minutes the sale will be available
     */
    public void putForAuction(int priceNQT, int timeout) {
       if (highestBidder==null && owner.equals(this.getCurrentTxSender())) {
          // only if there is no bidder and it is the current owner
          status = STATUS_FOR_AUCTION;
          auctionTimeout = getBlockTimestamp().addMinutes(timeout);
          currentPrice = priceNQT;
          sendMessage(status, owner.getId(), currentPrice,
auctionTimeout.getValue(), tracker);
       }
    }

    /**
     * If this contract is for sale or for auction, this method handles the
payment/bid.
     *
     * A buyer needs to transfer the asked price to the contract.
     *
     * If the token is for auction, a bidder needs to transfer more than the
current highest
     * bid to become the new highest bidder. Previous highest bidder is then
refunded (minus
     * the contract fee). After the auction timeout, any transaction received
```

```
will trigger
     * the ownership transfer.
     *
     * If the token was not for sale or the amount is not enough, the order is
     * refunded (minus the contract fees).
     */
    public void txReceived() {
       if (status == STATUS_FOR_SALE) {
          if (getCurrentTxAmount() >= currentPrice) {
             // Conditions match, let's execute the sale
             pay(); // pay the current owner
             owner = getCurrentTxSender(); // new owner
             status = STATUS_NOT_FOR_SALE;
             sendMessage(NEW_OWNER, owner.getId(), getCurrentTxAmount(), ZERO,
tracker);
             return;
          }
       }
       if (status == STATUS_FOR_AUCTION) {
          if (getBlockTimestamp().ge(auctionTimeout)) {
             // auction timed out, apply the transfer if any
             if (highestBidder != null) {
                pay(); // pay the current owner
                owner = highestBidder; // new owner
                highestBidder = null;
                status = STATUS_NOT_FOR_SALE;
                sendMessage(NEW_OWNER, owner.getId(), currentPrice, ZERO,
tracker);
             }
             // current transaction will be refunded below
          } else if (getCurrentTxAmount() > currentPrice) {
             // Conditions match, let's register the bid

             // refund previous bidder, if some
             if (highestBidder != null)
                sendAmount(currentPrice, highestBidder);

             highestBidder = getCurrentTxSender();
             currentPrice = getCurrentTxAmount();
             totalBidsReceived++;
             sendMessage(NEW_AUCTION_BID, highestBidder.getId(), currentPrice,
ZERO, tracker);
             return;
          }
       }
       // send back funds of an invalid order
       sendAmount(getCurrentTxAmount(), getCurrentTxSender());
    }

    private void pay() {
       amountToPlatform = currentPrice * platformFee / THOUSAND;
       amountToRoyalties = currentPrice * royaltiesFee / THOUSAND;

       totalPlatformFee += amountToPlatform;
       totalRoyaltiesFee += amountToRoyalties;
       totalTimesSold++;

       sendMessage(STATUS_NOT_FOR_SALE, owner.getId(), currentPrice, ZERO,
tracker);
       sendAmount(amountToRoyalties, royaltiesOwner);
       sendAmount(currentPrice - amountToPlatform - amountToRoyalties, owner);
    }

    protected void blockFinished(){
```

```
        // The platform fee is the remaining balance
        if(status == STATUS_NOT_FOR_SALE)
            sendBalance(platformAddress);
    }

    /**
     * A main function for debugging purposes only.
     *
     * This function is not compiled into bytecode and do not go to the
blockchain.
     */
    public static void main(String[] args) throws Exception {
        BT.activateCIP20(true);

        // some initialization code to make things easier to debug
        Emulator emu = Emulator.getInstance();
        Address creator = Emulator.getInstance().getAddress("CREATOR");
        emu.airDrop(creator, 1000 * Contract.ONE_BURST);

        Address token1 = Emulator.getInstance().getAddress("TOKEN");
        emu.createConctract(creator, token1, SignumArt.class,
Contract.ONE_BURST);

        emu.forgeBlock();

        new EmulatorWindow(SignumArt.class);
    }
}
```

ДОДАТОК Б

ЛІСТИНІНГ КОДУ CRYPTOBALL

```
public class Cryptoball extends Contract {

    public static final long CHECK_POWERPLAY10 = 10_000_000 * ONE_BURST;
    public static final long GAS_FEE = 2 * ONE_BURST;
    public static final long POWERPLAY = 1 * ONE_BURST;
    Timestamp lastRunning;
    long nextRoundBlock;
    Address holdersContract;
    Transaction nextTx;
    Timestamp lastPaid;
    Address nextToPay;
    long betnumbers;
    // Winning Balls
    long ball1,ball2,ball3,ball4,ball5;
    long cryptoball;
    long powerplay;
    //Helper for drawing
    long hash,hash1,hashBase;
    long hashCounter;
    long roundNumber;
    long drawNumber;
     // Bet number
    long playBall1,playBall2,playBall3,playBall4,playBall5;
    long playCryptoball;
    long playMultiPlay;
    long playPowerplay;
    //Helper for Bets
    long playcheck;
    long checkPlayNumber;
    long winningAmount;
    long playWinsCryptoball;
    long prize;
    long maxPrize;
    long ballsHit;
    // Number of blocks used to get the draw numbers hash
    long HASH_BLOCKS = 9;
    //static numbers
    long ZERO = 0;
    long ONE = 1;
    long TWO = 2;
    long THREE = 3;
    long FOUR = 4;
    long FIVE = 5;
    long SIX = 6;
    long SEVEN =7;
    long EIGHT = 8;
    long NINE = 9;
    long TEN =10;
    long HUNDRED =100;
    long TEN_THOUSAND =100;
    long FIFTY_THOUSAND = 50_000;
    long ONE_MILLION = 1_000_000;
    long TWENTY_SIX =26;
    long DISTRIBUTION_ROUNDS = 7;
    long SIXTY_NINE = 69;

    // time difference between rounds in minutes (24h = 1440 minutes)
    public static final long DELTA_T = 1440;
    long DELTA_T_BLOCKS = DELTA_T/4;
    long DELTA_LAST = HASH_BLOCKS*4;

    static final long HASH_MASK = 0x0FFFFFFFFFFFFFFFL;

    public Cryptoball(){
```

```
        holdersContract = getCreator();
        lastRunning = getBlockTimestamp().addMinutes(DELTA_T - DELTA_LAST);
        nextRoundBlock = getBlockHeight() + DELTA_T_BLOCKS - HASH_BLOCKS;
        // Infinite loop sorting at every DELTA_T blocks
       while(true) {
          // Avoid any delay if we need more than one block to run all the
bets
          sleep(nextRoundBlock - getBlockHeight());
            drawing();
            checkBets();
          nextRoundBlock = nextRoundBlock + DELTA_T_BLOCKS;
            lastRunning = lastRunning.addMinutes(DELTA_T);
            // ONE percent for holders on every week
            if(roundNumber % DISTRIBUTION_ROUNDS == ZERO) {
                //sendAmount(getCurrentBalance() * ONE / HUNDRED,TRT_Holders);
                sendAmount(getCurrentBalance() * ONE / HUNDRED,
this.holdersContract);
            }
       }
    }

    private void drawing(){
       // construct the winning seed based on 9 different blocks
        hash1 = getPrevBlockHash1();
        hash = hash1;
        hashCounter = ONE;
        while (hashCounter < HASH_BLOCKS){
            sleep(ONE);
            hash *= TWO; // shift by one bit
            hash += (getPrevBlockHash1() & ONE);
            hashCounter += ONE;
         }
         hash = performSHA256_64(hash, hash1);
         roundNumber += ONE;
         hashBase = hash;
         // Reset and Get all balls
         drawNumber = ZERO;
         ball1 = ZERO;
         ball2 = ZERO;
         ball3 = ZERO;
         ball4 = ZERO;
         ball5 = ZERO;
         ball1 = drawNumber();
         ball2 = drawNumber();
         ball3 = drawNumber();
         ball4 = drawNumber();
         ball5 = drawNumber();
         // Cryptoball
         hash = performSHA256_64(hash, hash);
         hash1 = hash & HASH_MASK;
         cryptoball = hash1 % TWENTY_SIX;
         cryptoball = cryptoball + ONE;
         // Powerplay
         powerplay = (hash1/TEN_THOUSAND) % FOUR;
         powerplay +=TWO;
         if(getCurrentBalance() > CHECK_POWERPLAY10 && powerplay == FIVE ){
             powerplay = TEN;
         }
        //Announce the winning seed (all numbers derived form it)
        sendMessage(hashBase, powerplay, this.holdersContract);
    }

    private long drawNumber(){
        while(drawNumber==ball1 || drawNumber==ball2 || drawNumber==ball3 ||
```

```
drawNumber==ball4 || drawNumber==ball5){
          hash = performSHA256_64(hash, hash);
          drawNumber =  hash & HASH_MASK;
          drawNumber = drawNumber % SIXTY_NINE;
            drawNumber +=ONE;
        }
       return drawNumber;
    }

    private void checkBets(){
        while (true) {
            nextTx = getTxAfterTimestamp(lastPaid);
          if (nextTx == null || nextTx.getTimestamp().ge(lastRunning))
             break;
            nextToPay = nextTx.getSenderAddress();
            betnumbers = nextTx.getMessage1();
            playcheck = checkPlay();
            lastPaid = nextTx.getTimestamp();
            if (playcheck == ONE){
                playWinsCryptoball =ZERO;
                if (playCryptoball == cryptoball){
                    playWinsCryptoball = ONE;
                }
                // Check the number of balls hit
              ballsHit = ZERO;
              checkBallHit(playBall1);
              checkBallHit(playBall2);
              checkBallHit(playBall3);
              checkBallHit(playBall4);
                checkBallHit(playBall5);
                winningAmount = ZERO;
                prize = ZERO;
                if (ballsHit == FIVE){
                    prize = (ONE_MILLION + (playPowerplay * ONE_MILLION)) *
ONE_BURST;
                }
                else if (ballsHit + playWinsCryptoball == FIVE){
                    winningAmount = FIFTY_THOUSAND;
                }
                else if (ballsHit + playWinsCryptoball == FOUR){
                    winningAmount = TEN_THOUSAND;
                }
                else if (ballsHit + playWinsCryptoball == THREE){
                    winningAmount = SEVEN;
                }
                else if (playWinsCryptoball == ONE){
                    winningAmount = FOUR;
                }
                if(winningAmount > ZERO && winningAmount <= FIFTY_THOUSAND){
                    prize = winningAmount * playMultiPlay;
                    prize += winningAmount * playPowerplay * (powerplay-ONE) *
playMultiPlay;
                    prize = prize * ONE_BURST;
                }
                maxPrize = getCurrentBalance() / TWO;
                if (ballsHit + playWinsCryptoball == SIX){
                    prize = maxPrize;
                }
                else if (prize > maxPrize){
                    prize = maxPrize;
                }
                if(prize > ZERO){
                    sendAmount(prize, nextTx.getSenderAddress());
                }
```

```
            }
        }
    }

    private long checkPlay(){
        // check Powerplay
        playPowerplay = betnumbers % TEN;
        if (playPowerplay != ONE && playPowerplay != ZERO){
           return ZERO;
        }
        betnumbers = betnumbers /TEN;
        // set  Multiplay by Amount send to contract
        playMultiPlay = (nextTx.getAmount() + GAS_FEE)/( GAS_FEE +
(playPowerplay * POWERPLAY));
        // check Cryptoball
        playCryptoball = betnumbers % HUNDRED;
        if (playCryptoball < ONE  && playCryptoball > TWENTY_SIX){
            return ZERO;
        }
        //Check Balls 5 to 1
        playBall5 = getPlayNumber();
        playBall4 = getPlayNumber();
        playBall3 = getPlayNumber();
        playBall2 = getPlayNumber();
        playBall1 = getPlayNumber();
        //Check all Balls 1 to 5 are unique
        if(playBall1 == playBall2 || playBall1 == playBall3 || playBall1 ==
playBall4 || playBall1 == playBall5
                || playBall2 == playBall3 || playBall2 == playBall4 ||
playBall2 == playBall5
                || playBall3 == playBall4 || playBall3 == playBall5
                || playBall4 == playBall5){
            return ZERO;
        }
        return ONE;
    }

    private long getPlayNumber(){
        betnumbers = betnumbers /HUNDRED;
        checkPlayNumber = betnumbers % HUNDRED;
        return checkPlayNumber;
    }

    private void checkBallHit(long playerBall){
       if (playerBall == ball1 ||  playerBall == ball2 ||  playerBall == ball3
||  playerBall == ball4 ||  playerBall == ball5){
          ballsHit += ONE;
       }
    }

    public void txReceived(){
        // Not needed, since the contract never leaves the loop on
constructor.
    }

    public static void main(String[] args) {
        BT.activateCIP20(true);
        new EmulatorWindow(Cryptoball.class);
    }
}
```

ДОДАТОК В

ЛІСТИНІНГ КОДУ СМАРТ-КОНТРАКТУ

```
public abstract class Contract {

    public final static long ONE_BURST = 100000000L;
    public final static long FEE_QUANT = 735000L;
    public final static long STEP_FEE = FEE_QUANT / 10L; // After AT2 fork,
othewise ONE BURST/10

    Address address;
    Address creator;
    Timestamp creation;

    Transaction currentTx;
    long activationFee;

    // Thread stuff to emulate sleep functions
    Semaphore semaphore = new Semaphore(1);
    boolean running;
    Timestamp sleepUntil;

    protected Contract() {
       Emulator emu = Emulator.getInstance();
       setInitialVars(emu.curTx, new
Timestamp(emu.getCurrentBlock().getHeight(), 0));
    }

    /**
     * Utility function that return the address of a given BURST-account
     *
     * @param account
     * @return
     */
    protected Address parseAddress(String rs) {
       return Emulator.getInstance().getAddress(rs);
    }

    /**
     * Utility function that return the address of a given ID
     * @param id the signed long id
     * @return the address
     */
    protected Address getAddress(long id) {
       return
Emulator.getInstance().getAddress(SignumCrypto.getInstance().rsEncode(SignumID
.fromLong(id)));
    }

    /**
     * Send the entire balance to the given address.
     *
     * Care should be exercised with this function since a contract with no
balance
     * cannot continue to run!
     *
     * @param ad the address
     */
    protected void sendBalance(Address ad) {
       sendAmount(this.address.balance, ad);
    }

    /**
     * Send the given amount to the receiver address
     *
     * @param amount
     * @param receiver
```

```
     */
    protected void sendAmount(long amount, Address receiver) {
       Emulator.getInstance().send(address, receiver, amount);
    }

    /**
     * Send the given message to the given address.
     *
     * The message has the isText flag set as false, the hexadecimal values
are
     * converted to characters when shown in BRS wallet. Message is always
     * unencrypted.
     *
     * @param message  the message, truncated in 4*sizeof(long)
     * @param amount   the amount or 0 to send the message only
     * @param receiver the address
     */
    protected void sendMessage(String message, Address receiver) {
       Emulator.getInstance().send(address, receiver, 0, message);
    }

    /**
     * Send the given message to the given address.
     *
     * The message has the isText flag set as false, the hexadecimal values
are
     * converted to characters when shown in BRS wallet. Message is always
     * unencrypted.
     *
     * @param message  the message in form of a 4 longs register
     * @param receiver the address
     */
    protected void sendMessage(Register message, Address receiver) {
       Emulator.getInstance().send(address, receiver, 0, message);
    }

    /**
     * Send the given message to the given address.
     *
     * The message has the isText flag set as false, the given hexadecimal
value is
     * converted to characters when shown in BRS wallet. Message is always
     * unencrypted.
     *
     * @param message  the message in form of a long number
     * @param receiver the address
     */
    protected void sendMessage(long message, Address receiver) {
       Emulator.getInstance().send(address, receiver, 0,
Register.newInstance(message, 0, 0, 0));
    }

    /**
     * Send the given message to the given address.
     *
     * The message has the isText flag set as false, the given hexadecimal
values are
     * converted to characters when shown in BRS wallet. Message is always
     * unencrypted.
     *
     * @param message  the message in form of a long number
     * @param message2  the message in form of a long number
     * @param receiver the address
     */
```

```
    protected void sendMessage(long message, long message2, Address receiver)
{
       Emulator.getInstance().send(address, receiver, 0,
Register.newInstance(message, message2, 0, 0));
    }

    /**
     * Send the given message to the given address.
     *
     * The message has the isText flag set as false, the given hexadecimal
values are
     * converted to characters when shown in BRS wallet. Message is always
     * unencrypted.
     *
     * @param message  the message in form of a long number
     * @param message2  the message in form of a long number
     * @param message3  the message in form of a long number
     * @param message4  the message in form of a long number
     * @param receiver the address
     */
    protected void sendMessage(long message, long message2, long message3,
long message4, Address receiver) {
       Emulator.getInstance().send(address, receiver, 0,
Register.newInstance(message, message2, message3, message4));
    }

    /**
     * Function to be called before processing the transactions of the current
     * block.
     *
     * Users should overload this function if there is some action to be taken
     * before processing the first {@link #txReceived()} for the current
block.
     *
     * Use this function carefully, since #{@link #getCurrentTx()} and similar
     * functions are still unavalable inside this function.
     */
    protected void blockStarted() {
    }

    /**
     * Function to be called when all transactions on the current block were
     * processed.
     *
     * Users should overload this function if there is some action to be taken
after
     * the last {@link #txReceived()} was called for the current block.
     */
    protected void blockFinished() {
    }

    /**
     * Get the first transaction received after the given timestamp
     *
     * @param ts
     * @return
     */
    protected Transaction getTxAfterTimestamp(Timestamp ts) {
       return Emulator.getInstance().getTxAfter(address, ts);
    }

    /**
     * @return the transaction being processed at this moment
     */
```

```
    protected Transaction getCurrentTx() {
        return currentTx;
    }

    /**
     * @return the current transaction timestamp
     */
    protected Timestamp getCurrentTxTimestamp() {
        return getCurrentTx().getTimestamp();
    }

    /**
     * @return the current transaction sender address
     */
    protected Address getCurrentTxSender() {
        return getCurrentTx().getSenderAddress();
    }

    /**
     * @return the current transaction amount
     */
    protected long getCurrentTxAmount() {
        return getCurrentTx().getAmount();
    }

    /**
     * @return the block hash of the previous block (part 1 of 4)
     */
    protected Register getPrevBlockHash() {
        return Emulator.getInstance().getPrevBlock().hash;
    }

    /**
     * @return the first part of the previous block hash
     */
    protected long getPrevBlockHash1() {
        return Emulator.getInstance().getPrevBlock().hash.getValue1();
    }

    /**
     * @return the timestamp of the previous block
     */
    protected Timestamp getPrevBlockTimestamp() {
        return new Timestamp(Emulator.getInstance().getPrevBlock().getHeight(),
0);
    }

    /**
     * @return the timestamp of the block being processed
     */
    protected Timestamp getBlockTimestamp() {
        return new
Timestamp(Emulator.getInstance().getCurrentBlock().getHeight(), 0);
    }

    /**
     * @return the timestamp of the block being processed
     */
    protected long getBlockHeight() {
        return Emulator.getInstance().getCurrentBlock().getHeight();
    }

    /**
     * @return the address of the creator of this contract
```

```
     */
    protected Address getCreator() {
       return creator;
    }

    /**
     * @return the creation timestamp of this contract
     */
    protected Timestamp getCreationTimestamp() {
       return creation;
    }

    /**
     * @return the current balance of this contract
     */
    protected long getCurrentBalance() {
       return address.balance;
    }

    @EmulatorWarning
    public static Register performSHA256_(Register input) {
       Register ret = new Register();

       ByteBuffer b = ByteBuffer.allocate(32);
       b.order(ByteOrder.LITTLE_ENDIAN);

       for (int i = 0; i < 4; i++) {
          b.putLong(input.value[i]);
       }

       try {
          MessageDigest sha256 = MessageDigest.getInstance("SHA-256");
          ByteBuffer shab = ByteBuffer.wrap(sha256.digest(b.array()));
          shab.order(ByteOrder.LITTLE_ENDIAN);

          for (int i = 0; i < 4; i++) {
             ret.value[i] = shab.getLong(i * 8);
          }
       } catch (NoSuchAlgorithmException e) {
          // not expected to reach that point
          e.printStackTrace();
       }
       return ret;
    }

    /**
     * @return a SHA256 hash of the given input
     */
    protected Register performSHA256(Register input) {
       return performSHA256_(input);
    }

    /**
     * A utility function returning only the first 64 bits of a SHA256 for two
long
     * inputs.
     *
     * @return the first 64 bits SHA256 hash of the given input
     */
    protected long performSHA256_64(long input1, long input2) {
       Register input = new Register();
       input.value[0] = input1;
       input.value[1] = input2;
```

```
        Register ret = performSHA256_(input);
        return ret.getValue1();
    }

    /**
     * Sleeps until the contract receives a new transaction.
     */
    protected void sleepUntilNextTx() {
        // FIXME: on emulator we will sleep for one block
        sleep(0);
    }

    /**
     * Sleeps for the given number of blocks.
     *
     * @param nblocks number of blocks to sleep
     */
    protected void sleep(long nblocks) {
        if(nblocks <= 0)
            sleepUntil = null;
        else {
            sleepUntil = new
Timestamp(Emulator.getInstance().getCurrentBlock().height + nblocks, 0);
            address.setSleeping(true);
            try {
                semaphore.acquire();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        address.setSleeping(false);
        sleepUntil = null;
    }

    /**
     * A new transaction was received.
     *
     * Overload this function to implement your contract.
     */
    public abstract void txReceived();

    // ============================================================
    // Functions not visible, will be used by the virtual
    // machine when in debug mode but not exported to the
    // AT blockchain machine code

    void setInitialVars(Transaction tx, Timestamp creation) {
        this.creator = tx.sender;
        this.address = tx.receiver;
        this.creation = creation;
        this.activationFee = tx.getAmount();
        this.address.contract = this;

        // we acquire the semaphore in the creation process, it is released
        // when finished by the emulator
        try {
            running = true;
            semaphore.acquire();
        } catch (InterruptedException e) {
            running =false;
        }
    }

    void setCurrentTx(Transaction current) {
```

```
        this.currentTx = current;
    }

    @EmulatorWarning
    public String getFieldValues() {
        String ret = "<html>";
        Field[] fields = this.getClass().getDeclaredFields();
        for (Field f : fields) {
            try {
                f.setAccessible(true);
                Object v = f.get(this);
                ret += "<b>" + f.getName() + "</b> = " + (v != null ?
v.toString() : "null") + "<br>";
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return ret;
    }
}
```

ДОДАТОК Г

ЛІСТИНІНГ КОДУ ПРИКЛАДУ NFT2

```
@TargetCompilerVersion(CompilerVersion.v0_0_0)
public class NFT2 extends Contract {

    public static final int STATUS_NOT_FOR_SALE = 0;
    public static final int STATUS_FOR_SALE = 1;
    public static final int STATUS_FOR_AUCTION = 2;

    public static final long ACTIVATION_FEE = ONE_BURST * 1;

    int status;
    Address owner;

    long salePrice;
    Timestamp auctionTimeout;
    Address highestBidder;
    long highestBid;

    /**
     * Constructor, when in blockchain the constructor is called when the
first TX
     * reaches the contract.
     */
    public NFT2() {
       // Initially token is owned by the creator
       owner = getCreator();
       status = STATUS_NOT_FOR_SALE;
    }

    /**
     * Transfers the ownership of this token.
     *
     * Only the current owner can transfer the ownership.
     *
     * @param newOwner
     */
    public void transfer(Address newOwner) {
       if (highestBidder==null && owner.equals(this.getCurrentTxSender())) {
          // only if there is no bidder and it is the current owner
          owner = newOwner;
          status = STATUS_NOT_FOR_SALE;
       }
    }

    /**
     * Cancels an open for sale or auction and sets it as not for sale.
     */
    public void setNotForSale(){
       if (highestBidder==null && owner.equals(this.getCurrentTxSender())) {
          // only if there is no bidder and it is the current owner
          status = STATUS_NOT_FOR_SALE;
          salePrice = 0;
       }
    }

    /**
     * Put this token for sale for the given price.
     *
     * Buyer needs to transfer at least the asked amount.
     *
     * @param priceNQT the price in NQT==1E-8 BURST (buyer needs to transfer
at least
     *                 this amount)
     */
    public void putForSale(long priceNQT) {
```

```
        if (highestBidder==null && owner.equals(this.getCurrentTxSender())) {
            // only if there is no bidder and it is the current owner
            status = STATUS_FOR_SALE;
            salePrice = priceNQT - ACTIVATION_FEE;
        }
    }

    /**
     * Put this token for auction with the minimum bid price.
     *
     * Bidders need to transfer more than current highest to become the new
     * highest bidder. Previous highest bidder is refunded in case of a new
     * highest bid arrives.
     *
     * @param priceNQT the minimum bid price in NQT==1E-8 BURST (buyer need to
     *                 transfer at least this amount)
     * @param timeout  how many minutes the sale will be available
     */
    public void putForAuction(int priceNQT, int timeout) {
        if (highestBidder==null && owner.equals(this.getCurrentTxSender())) {
            // only if there is no bidder and it is the current owner
            status = STATUS_FOR_AUCTION;
            auctionTimeout = getBlockTimestamp().addMinutes(timeout);
            highestBid = priceNQT - ACTIVATION_FEE;
        }
    }

    /**
     * If this contract is for sale or for auction, this method handles the
payment/bid.
     *
     * A buyer needs to transfer the asked price to the contract.
     *
     * If the token is for auction, a bidder need to transfer more than the
current highest
     * bid to become the new highest bidder. Previous highest bidder is then
refunded (minus
     * the activation fee). After the auction timeout, any transaction
received will trigger
     * the ownership transfer.
     *
     * If the token was not for sale or the amount is not enough, the order is
     * refunded (minus the contract activation fee).
     */
    public void txReceived() {
        if (status == STATUS_FOR_SALE) {
            if (getCurrentTxAmount() >= salePrice) {
                // Conditions match, let's execute the sale
                sendAmount(salePrice, owner); // pay the current owner
                owner = getCurrentTxSender(); // new owner
                status = STATUS_NOT_FOR_SALE;
                return;
            }
        }
        if (status == STATUS_FOR_AUCTION) {
            if (getBlockTimestamp().ge(auctionTimeout)) {
                // auction timed out, apply the transfer if any
                if (highestBidder != null) {
                    sendAmount(highestBid, owner); // pay the current owner
                    owner = highestBidder; // new owner
                    highestBidder = null;
                    status = STATUS_NOT_FOR_SALE;
                }
                // current transaction will be refunded below
```

```
            } else if (getCurrentTxAmount() > highestBid) {
                // Conditions match, let's register the bid

                // refund previous bidder, if some
                if (highestBidder != null)
                    sendAmount(highestBid, highestBidder);

                highestBidder = getCurrentTxSender();
                highestBid = getCurrentTxAmount();
                return;
            }
        }
        // send back funds of an invalid order
        refund();
    }

    /**
     * Send back funds of the current transaction.
     */
    void refund() {
        // send back funds of an invalid order
        sendAmount(getCurrentTxAmount(), getCurrentTxSender());
    }

    /**
     * A main function for debugging purposes only.
     *
     * This function is not compiled into bytecode and do not go to the
blockchain.
     */
    public static void main(String[] args) throws Exception {
        BT.activateCIP20(true);

        // some initialization code to make things easier to debug
        Emulator emu = Emulator.getInstance();
        Address creator = Emulator.getInstance().getAddress("CREATOR");
        emu.airDrop(creator, 1000 * Contract.ONE_BURST);

        Address token1 = Emulator.getInstance().getAddress("TOKEN");
        emu.createConctract(creator, token1, NFT2.class, Contract.ONE_BURST);

        emu.forgeBlock();

        new EmulatorWindow(NFT2.class);
    }
}
```